Addax: A fast, private, and accountable ad exchange infrastructure

Ke Zhong, Yiping Ma, and Yifeng Mao, *University of Pennsylvania;*
Sebastian Angel, *University of Pennsylvania & Microsoft Research*

https://www.usenix.org/conference/nsdi23/presentation/zhong

This paper is included in the
Proceedings of the 20th USENIX Symposium on
Networked Systems Design and Implementation.

April 17–19, 2023 • Boston, MA, USA

978-1-939133-33-5

Addax: A fast, private, and accountable ad exchange infrastructure

Ke Zhong⋆ Yiping Ma⋆ Yifeng Mao⋆ Sebastian Angel⋆†
⋆*University of Pennsylvania* †*Microsoft Research*

Abstract. This paper proposes Addax, a fast, verifiable, and private online ad exchange. When a user visits an ad-supported site, Addax runs an auction similar to those of leading exchanges; Addax requests bids, selects the winner, collects payment, and displays the ad to the user. A key distinction is that bids in Addax's auctions are kept private and the outcome of the auction is publicly verifiable. Addax achieves these properties by adding public verifiability to the affine aggregatable encodings in Prio (NSDI'17) and by building an auction protocol out of them. Our implementation of Addax over WAN with hundreds of bidders can run roughly half the auctions per second as a non-private and non-verifiable exchange, while delivering ads to users in under 600 ms with little additional bandwidth requirements. This efficiency makes Addax the first architecture capable of bringing transparency to this otherwise opaque ecosystem.

1 Introduction

Ad exchanges such as DoubleClick and OpenX are key players in online advertising; their role is to auction ad space on a publisher's website in real time to advertisers. When a user visits a publisher's page, the user's browser contacts a server that triggers an auction on an exchange. The exchange gives advertisers information about the publisher (e.g., URL, ad size and type, category of site) and the user (e.g., demographic, metadata for syncing cookies across sites) in real time, and collects bids from interested parties. The exchange then runs an auction (e.g., second-price auction), delivers to the user the ads of the winning advertisers, and credits the publisher. Finally, technologies like header bidding [4] and Google's open bidding platform [30] allow publishers to auction users across many exchanges (essentially an exchange of exchanges), increasing competition and improving publishers' revenue.

While ad networks and exchanges serve as the financial backbone of the free web, their centralized nature means that: (1) they are privy to sensitive information, including user's browsing habits and the preferences and valuations of advertisers; and (2) they are opaque and hard to audit. The former has received considerable attention [39, 45, 56, 71, 77, 89, 91]; the lack of auditing mechanisms and the knowledge of advertisers' valuations is becoming a serious sociotechnical issue. A recent antitrust lawsuit alleges that Google used insider knowledge of past bids submitted by advertisers to gain unfair advantages whenever its subsidiaries participated in auctions [63]. Further, it is alleged that Google convinced Facebook to not participate in header bidding—a technology considered an "existential threat" to Google's business [28, 81, 84]. According to disclosed reports, in return for Facebook choosing to participate instead in Google's open bidding platform, "Google provided Facebook with special information and speed advantages to help [Facebook's exchange] succeed in the auction [over other bidders]—even including a guaranteed win rate" [84].

Regardless of the merits of these cases, the key issue—and the crux of this paper—is that there are no ways for exchanges to *prove* to their customers and to regulators that they are not abusing their position. To address this, we present Addax, an online advertising architecture that achieves 4 goals:

- *Auction integrity.* Auctions should be publicly verifiable to allow the ad exchange to prove that it is not biasing auctions towards particular bidders or lying about their outcome.
- *Auction privacy.* The bids of losing bidders should be hidden from all parties—even the exchange itself! This ensures that the exchange cannot abuse or share this information.
- *High performance.* Addax should handle the stringent performance requirements of the ad ecosystem.
- *Better tracking.* Addax should work with recent tracking efforts such as Google's Topics API [16] and Microsoft's PARAKEET [13] that allow targeted ads but collect less information about individuals.

Overview. In Addax, browsers track users' histories with existing privacy-preserving client-side techniques [57, 58, 80, 89], and kickstart verifiable and private auctions whenever the user navigates to an ad-supported site. Auctions in Addax proceed in three steps. First, the browser invites relevant bidders (e.g., demand-side platforms) by finding their information (e.g., URL of their ad server) on a database. In existing header bidding platforms [2] such databases are currently maintained by publishers; Addax preserves this model, but we additionally experiment with a more decentralized approach where the database is maintained in a public append-only log and discuss how to reduce the cost of lookups in this model (§7). As part of the invitation, the browser supplies to bidders information about the site being visited and a variable amount of user information based on the user's configuration of Addax (ranging from fully targeted to generic ads).

Second, bidders submit encrypted bids to the publisher and one or more *auxiliary servers*. The auxiliary server helps the publisher run a new lightweight secure auction computation over the encrypted bids (§4). The role of an auxiliary server could be taken up by today's exchanges or it can be a separate entity propped up by the industry at large. Under the *anytrust model* [88] (either the publisher or any of the auxiliary servers is honest), the secure auction computation returns the winning bidder's identity and bid, and the auction's sale price, but no other information.

Last, Addax produces an *audit trail* that is uploaded to a public log and that allows any auditor to verify that the auction was conducted with integrity (§5). At the conclusion of the auction, the browser fetches the ad from the winning bidder (or a content distribution network) and the publisher learns which bidder and how much to bill for the ad impression.

Technical contributions. To maintain good performance, Addax cannot use expensive cryptography (e.g., homomorphic encryption, multiparty computation) in order to achieve the integrity and privacy goals. Indeed, our evaluation of such baselines confirms that they are far too inefficient to meet online advertising's low latency and communication requirements (§9). Instead, Addax makes three contributions:

Secure auction protocol. Addax introduces a new auction protocol based on Prio's *affine-aggregatable encodings* (AFE) [51]. Addax's auction is simple and lightweight and allows two or more parties to run the auction over secret-shared bids without revealing anything beyond the auction's outcome.

Verifiable AFEs (V-AFEs). Addax extends Prio's AFEs to provide public verifiability for *outputs* (Prio has mechanisms to verify inputs). Addax then uses V-AFEs to allow anyone (e.g., an auditor) to confirm that an auction was conducted correctly without learning any of the input bids.

Integration with Algorand and Chrome. Addax implements mechanisms to interact quickly with the public log (we use Algorand [3, 55]) and smart contracts to manage the registration of advertisers and the collection of audit materials. Addax also leverages Chrome native messaging to launch auctions.

Our implementation of Addax can complete auctions over WAN with twice the average number of bidders reported in production ad exchanges [92] in 440–580 ms (for first and second-price auctions), and requires only 1.2 MB of communication between the publisher and the auxiliary server (§9.2). This is fast enough for ads to be loaded asynchronously without affecting page load time for the overwhelming majority of websites today [1, 7, 20, 87]. In terms of throughput, Addax can handle around 250–360 auctions per second per core (for second and first-price auctions), which is roughly 40% of what our non-private and unverified baseline can achieve. Creating the audit trail requires additional computation on the part of bidders but adds negligible overhead to users' browsers and publishers. In contrast, the same auction implemented in existing state-of-the-art cryptographic frameworks (MPC and FHE) requires over 4 GB of communication and over 100 sec.

Limitations. Ad exchanges do more than just run auctions and deliver ads. They vet advertisers to ensure users do not receive malware; mitigate fraud; and provide powerful analytics. Addax does not yet address these complementary and critical aspects, but Section 11 discusses concrete directions to incorporate such features into Addax's architecture. Finally, Addax can achieve better performance (optionally) at the expense of revealing the existence of winning ties (§4.4).

2 Background and goals

Ad exchanges are platforms that auction *impressions* (the display of a text, image, or video ad) on a publisher's website or mobile application in real time. Exchanges support highly targeted advertising whereby bidders (advertisers or their representatives, called demand-side platforms) get a chance to evaluate the publisher and the user to whom the ad will be shown to decide how much they would be willing to pay (if at all). This type of programmatic *real-time bidding* (RTB) advertising accounts for over a third of all digital ad spending today [22, 27]. Some of the largest ad exchanges include DoubleClick, PubMatic, OpenX, and Facebook.

To participate in an ad exchange, a publisher inserts a *supply-side platform*'s (SSP) iframe or JavaScript snippet into their page. An SSP is a service that sells the publisher's ads on an exchange (publishers can also run their own SSP). When a user's browser fetches the publisher's site and executes the provided JavaScript, it sends an HTTP GET request to the SSP supplying the user's cookie, and awaiting for an ad to be returned. At this point, the SSP can identify the user and publisher, and start an RTB auction. During this process, the exchange invites dozens of potentially interested bidders to bid on the user [92], supplying them with demographic information, and relevant details about the publisher and the ad space (size, type, location within the page). To facilitate the valuation of the user, exchanges and bidders synchronize cookies [19, 23] to allow bidders to learn the identity of the user in their respective platforms (if applicable). Based on this information, bidders return a bid in CPM (cost per 1000 impressions), which ranges from cents to tens of dollars [32].

Upon receiving all bids, the exchange runs an auction where it selects the winning bidder and charges them the auction's sale price based on the type of auction. Two common types are *first-price* (winner pays what they bid) and *second-price* [83] (winner pays second highest bid) auctions. Finally, the exchange notifies the SSP with the result of the auction, who then responds to the user's GET requests with the information that the browser needs to retrieve the ads (images, videos, etc.) from a storage server.

2.1 Header bidding

Header bidding [4] is a recent advertising paradigm where the publisher (or its SSP) works with multiple exchanges to sell its ad slot in real time. It is called "header bidding" because the publisher supplies JavaScript code that runs in the `<header>` part of the page (which loads as soon as the page starts opening in the user's browser), and this code triggers the process of contacting the exchanges. The exchanges then internally run their own auctions (first or second-price) and send back the winning bids to the browser. The browser then sends the winning bids to the publisher (or its SSP), which runs another auction (typically first-price), selects the highest bid as the winner, and forwards the winner's ad tag to the user's browser. Google's Open Bidding platform is similar [30].

2.2 Concerns with existing exchanges

We highlight three areas of concern with existing ad exchanges. First, there is no visibility into the auction process. VEX [35] argues that this opens the door to a variety of issues—including those that are mentioned by the antitrust lawsuits [28, 63, 84]. Second, exchanges observe all submitted bids in the clear. These bids represent how valuable different users and publishers are to bidders, which reveals information about bidder's trading algorithms, finances, and future plans. Last, users lack agency and have no say over which types of ads they receive or what information is shared with bidders. One might imagine a different world in which users can express an opinion on the types of ads they consume (e.g., no ads for kids toys to avoid children exploitation), and what information about themselves they reveal in order to receive targeted ads.

2.3 Goals

Addax aims to address many of the shortcomings of existing ad exchanges by giving agency to users, privacy to bidders, and transparency to all. Addax is compatible with both traditional exchanges and with the header bidding model (including Google's open bidding platform). We detail these goals next.

Integrity of the auction. All parties should be able to verify that Addax's auctions are conducted correctly, as per the auction type (first-price, second-price, etc.).

Privacy for losing bids. Addax should hide the bids of all of the losing bidders from *everyone*, even the auctioneers. One exception is that in second-price auctions, the second highest bid (which is technically a losing bid) becomes the sale price and cannot be hidden.

Privacy among bidders. Bidders should not need to learn each others' identities or interact with one another in order to participate in an auction. Existing exchanges do not reveal this information, and neither should Addax.

High performance. Addax must ensure that auctions complete quickly, as ads need to be displayed within hundreds of milliseconds in order to preserve a good user experience and follow existing RTB requirements [10, 11].

User agency. Addax's focus is on making the auction process accountable without exposing bidders' information. Addax should also allow users to have a say on which kinds of ads they wish to receive. Ideally, Addax would also improve user privacy, but this is not a goal of this work. Instead, we ask that Addax make things no worse than they are today for users, and that it be compatible with other works that aim to reduce user tracking (such as Topics [16]). Appendix G expands on this compatibility aspect.

2.4 Potential solutions (baselines)

Given Addax's desire for privacy and verifiability, one might ask whether existing tools such as homomorphic encryption or multiparty computation fit the bill. This is not the case.

Homomorphic encryption (HE). HE libraries [24–26, 61] allow the computation of additions and multiplications over encrypted data without access to plaintext values. Computing an auction, however, requires comparisons (such as "less than") which are expensive to express with arithmetic operations as they typically require decomposing values into bits and encrypting bits separately [25, 50]. Even recent optimizations are expensive [41, 48, 64]. As we show in our evaluation (§9.2) an auction with 96 bidders using the state-of-the-art TFHE library [49, 50] takes 181 seconds. Finally, HE lacks integrity: an auctioneer is free to compute an incorrect auction. Recent work on composing verifiable computation with HE can address this, but at orders-of-magnitude cost increase [40, 54].

Secure multi-party computation (MPC). MPC frameworks [31, 65, 85] allow mutually distrusting parties to compute a function over secret inputs without revealing anything beyond the function's outcome. It might seem natural to encode the auction as an MPC among the bidders but this is impractical when there are many bidders. An alternative is to use a *delegated* MPC setting whereby two parties (publisher and auxiliary server in our setting) run the MPC on behalf of others; bidders could send secret shares of their bids to these two parties. However, this delegated setting lacks integrity: either party is free to supply bogus shares to the MPC to cause the auction's output to be *undetectably* incorrect. As we show in Section 9.2, addressing this introduces prohibitive costs.

Trusted execution environments (TEEs). Another possibility is to use trusted hardware. Besides side channel [43, 46, 68, 90] and integrity attacks [72, 82], TEEs alone cannot solve this problem. Appendix F discusses this in depth.

3 Addax Overview

Addax is a platform where the exchange's duties are split among different parties. Figure 1 gives a high-level description. Addax consists of: (i) publishers who run their own SSP and who wish to show ads to fund their services, (ii) the client's browser, (iii) auxiliary servers who help to run auctions, (iv) bidders (demand-side platforms, advertisers, other exchanges, etc.) who bid on ad slots, and (v) an append-only log (e.g., blockchain, BFT consortium) for persisting an audit trail. We discuss what happens when a user visits a page below, and give details in the sections that follow. We defer a discussion of how bidders join Addax and what information they supply to Section 7 and Appendix E.

Steps ①–②: Client visits a publisher. When a client visits a publisher, it receives the page content, along with a unique *auction id* and a list of valid ad categories that the publisher supports. Addax uses the 392 categories from the Internet Advertising Bureau's (IAB) contextual taxonomy [10], which include things like "Humor", "Nutrition", etc. This metadata is embedded within the header of the page, as in header bidding (§2.1). An Addax-enabled browser, hereafter named "browser", parses the web page and extracts this metadata.

Step ③: Advertising filtering. Addax adopts a client-based tracking approach inspired by Privad [58], Adnostic [80], and Google's recent FLoC proposal [89]. Briefly, the browser tracks which sites the user visits over time and generates a profile of the user's interests, which it stores locally in a SQLite database similar to how cookies are stored. After parsing a page's ad spot metadata, the browser combines the user's profile, the ad spot's categories supplied by the publisher, and disallowed categories previously flagged by the user through a local configuration (e.g., to prevent categories that target children). Based on the refined information, the browser fetches bidders' details from a bidder database. Addax supports two types of databases: an embedded database supplied by the publisher during Step ②; and a public database where bidder information is maintained on the public log (blockchain). The former is how header bidding works today while the latter option is more decentralized and gives users more agency over the ads they receive. We defer the details to Section 7.

Steps ④–⑥: Private, decentralized, and verifiable auction. The browser invites the k bidders from step ③ to an auction. To do so, it provides them with an *auction id* (unique identifier supplied by the publisher), information about the user, and information to contact the publisher and the auxiliary server. Bidders decide whether to join the auction; if so, they respond to the auxiliary server and publisher with their required materials. The auxiliary server and the publisher collaboratively run the auction and select the auction's winner, and the auction's sale price. Asynchronously, and off the critical path, all participants upload to the public append-only log materials needed for public auditing (§5).

Step ⑦: Notify winner and display the ad. After the auction concludes, the publisher and auxiliary server learn the outcome (but nothing else). The publisher notifies the winner and asks it for an ad tag and payment (e.g., a signed IOU). The publisher then forwards the ad tag to the browser so that it can fetch and display the ad on the designated ad spot.

Verification. Auditors can use the information on the public log to verify the auction's outcome. By default, they only learn whether the auction was correct and the number of bidders that participated. In case that verification fails, Addax helps narrow down which parties were faulty (§5.3).

3.1 Assumptions and threat model

Addax assumes an append-only log (blockchain, BFT, etc.) and an *anytrust* model [88] where either the publisher or the auxiliary server is honest. The parties may act as follows.

Bidders. Bidders who are invited to the auction can submit bogus bids and cryptographic material. We model bidders as *covert adversaries* [36] who can deviate from the protocol arbitrarily as long as their malicious actions cannot be detected. If detected, bidders can incur financial or legal penalties, and can be banned by publishers. Addax assumes at least 2 non-colluding losing bidders (otherwise information about losing bids can be inferred from the outcome).

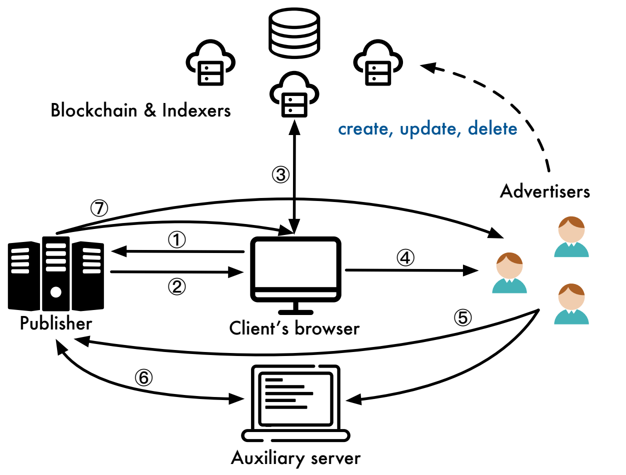


FIGURE 1—In Addax, the exchange's functionality is divided among the publisher, browser, an auxiliary server and a blockchain.

Publishers and auxiliary servers. Publishers may wish to increase their revenue by lying about the auction's outcome (e.g., forcing the winner to pay a fee higher than the second highest bid), learn the bids of losing bidders, or force users to view certain ads. Auxiliary servers may wish to bias the auction's result to help particular bidders. We model both parties as covert adversaries since detectable misbehavior can tarnish their reputation or incur legal penalties.

3.2 Security properties

Addax's auction protocol provides the following properties.

Completeness. If all parties are honest and the auction's outcome is correct (e.g., the winner is the highest bidder and the sale price is the second highest bid), then Addax's verification protocol passes with high probability.

Soundness. If a bidder, the publisher, or an auxiliary server misbehaves, Addax's verification fails with high probability.

Privacy. Addax's auction and verification hides all bids except the highest bid and the sale price.

4 Private ad auction

This section describes Addax's private ad auction. We begin by describing our building blocks.

4.1 Affine-aggregatable encodings (AFE)

Prio [51] shows how one can take two or more data values and encode each of them as a vector of λ bits such that adding up the vectors and running a decoding function on the sum is equivalent to computing some boolean function `f` (e.g., OR, AND, XOR) on the original data values; λ is a parameter that controls the probability of the result being correct. Prio calls this and other similar transformations an *Affine-Aggregatable Encoding* (AFE). Addax uses the "OR" boolean function to compute auctions, so it could use Prio's AFE. However, we

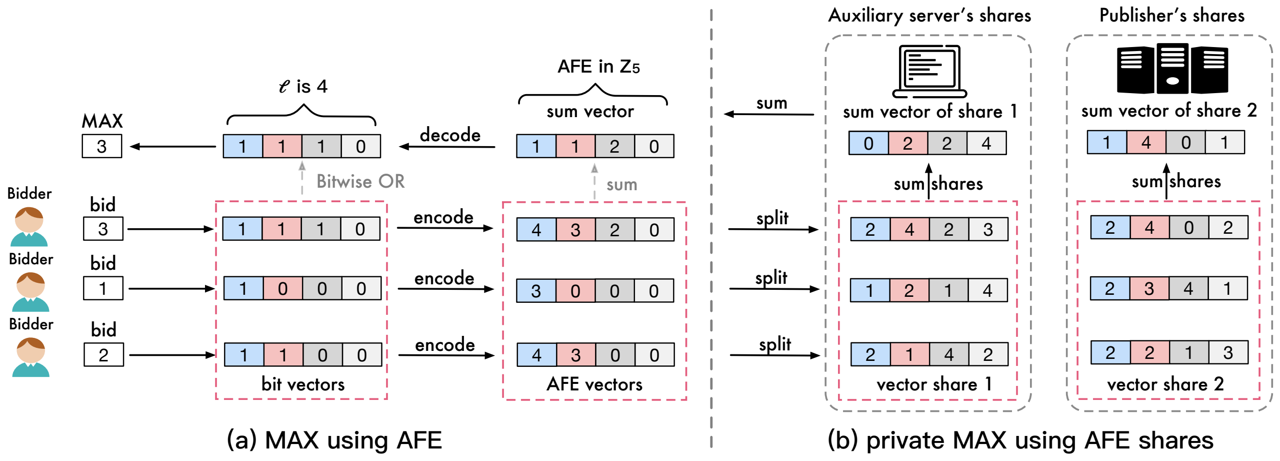


FIGURE 2—Example of (a) the MAX algorithm using AFE, and (b) the private MAX algorithm using AFE shares. In this example, $\ell = 4$ (which affects the range of bids) and we use AFE in $\mathbb{Z}_5$ (which affects the probability of obtaining the correct result).

depart slightly from Prio by encoding data values into a single element in $\mathbb{Z}_p$ (the set of integers modulo a large prime p) rather than λ-sized bit vectors. This encoding is more expensive than Prio's (since $\lambda < \log p$), but it allows Addax to add public verifiability, as we discuss in Section 5.1. Below we give our AFE for the "OR" function over bit values.

Encode OR. Given a bit $x \in \{0, 1\}$, its AFE is:

$$\text{Encode-OR}(x) = \begin{cases} 0 \in \mathbb{Z}_p & \text{if } x = 0 \\ \text{a random element} \in \mathbb{Z}_p & \text{if } x = 1 \end{cases}$$

Compute OR. Given a set of n AFE values $\{v_1, \ldots, v_n\}$, which encode n bits $\{x_1, \ldots, x_n\}$ with the above Encode-OR procedure, one can compute the OR of the n bits as:

$$\mathbf{v} = v_1 + \cdots + v_n \in \mathbb{Z}_p$$

Decode OR. Given the sum AFE value $\mathbf{v}$, one can correctly recover the result of the OR operation over the underlying n bits with probability of at least $1 - 1/p$ as follows.

$$\text{Decode-OR}(\mathbf{v}) = \begin{cases} 0 & \text{if } \mathbf{v} = 0 \\ 1 & \text{otherwise} \end{cases}$$

To see why Decode-OR returns the correct value with probability $1 - 1/p$, we consider two cases. First, when all n input bits are 0. In this case, all AFE values are zeros so $\mathbf{v}$ is guaranteed to be zero; Decode-OR always outputs the correct value of 0. Second, when at least one of the n input bits is 1. In this case, since the value is independent and uniformly random, the probability that the sum in $\mathbb{Z}_p$ is zero is $1/p$.

4.2 Computing the MAX function with AFE

Following the approach in Prio, we show how to extend the above AFE to support *MAX*, which Addax uses to find the highest bid in an auction. This construction provides neither privacy nor verifiability; we add these later.

Suppose all input values are integers in the range $[0, \ell]$. Each input x is first represented in unary. That is, as a bit vector β of length ℓ $(\beta_1, \beta_2, \ldots, \beta_\ell)$ where $\beta_i = 1$ if and only if $i \le x$. Observe that if we perform a bitwise OR on the unary bit vectors of all inputs, the result will be a unary bit vector where the index of the last "1" represents the maximum value across all inputs. This is the idea behind the AFE of MAX.

Encode MAX. Given a value $x \in [0, \ell]$, its AFE is a vector of ℓ values, where each value is an element in $\mathbb{Z}_p$. The encoding happens in two steps: (1) represent x as a bit vector β of length ℓ in unary format; and (2) for each bit β_i, encode β_i with the Encode-OR of Section 4.1. The result is a vector M with ℓ values, where $M[i]$ is the AFE value of bit β_i.

Compute MAX. Given n AFE vectors $\{M_1, \ldots, M_n\}$ that encode the values $\{x_1, \ldots, x_n\}$ as above, MAX is computed by adding the n vectors: $\mathbf{M} = M_1 + \cdots + M_n$.

Decode MAX. Given the sum vector $\mathbf{M}$, one can recover the MAX of the underlying n values in two steps. First, use Decode-OR (§4.1) on each of the ℓ entries of $\mathbf{M}$. The result is a bit vector β of length ℓ in unary format. Second, output the highest index j for which β_j is 1. This value is the correct MAX among the n inputs if Decode-OR outputs the correct OR for all ℓ bits. This event occurs with probability $\ge (1 - 1/p)^\ell$.

Figure 2(a) gives an example of these procedures with three inputs. Below we describe how to add privacy by secret sharing the AFE vectors among multiple parties.

4.3 Private and decentralized MAX

Observe that computing the MAX of n values $(x_1, \ldots, x_n)$ using AFE vectors $(M_1, \ldots, M_n)$ requires only additions. We can split each vector M_i into two shares (M_i^1 and M_i^2) that add up to the original ($M_i = M_i^1 + M_i^2$) as depicted in Figure 2(b). Each share is made up of uniform random elements in $\mathbb{Z}_p$, and reveals no information about M_i without the other share.

Suppose that two non-colluding parties, Alice and Bob, are tasked with computing the MAX of n values given n AFE vector shares. Alice receives $\{M_1^1, \ldots, M_n^1\}$ and Bob receives $\{M_1^2, \ldots, M_n^2\}$. Each party sums up their n shares to get two sum vector shares: M^1 for Alice and M^2 for Bob. Finally, both parties exchange their sum vector shares. Observe that by adding M^1 and M^2, the parties can recover the sum vector $\mathbf{M} = M^1 + M^2 = M_1 + \cdots + M_n$ as shown in Figure 2(a), and then use Decode-MAX to recover the max value.

4.4 Private and decentralized auction

We use the private MAX of Section 4.3 to compute an auction where the auctioneer's duty is split between the auxiliary server and the publisher. This protocol provides privacy but not integrity (i.e., malicious actions can lead to an incorrect outcome); we add verifiability in Section 5. A bid is given by the position of the last "1" in a unary vector (e.g., [1,1,1,0] and [0,0,1,0] both represent 3, though the latter is ill-formed). We assume a maximum bid ℓ, and everyone bids within $[0, \ell]$.

Step 1: Set up shared secret. Before the auction starts, the publisher and the auxiliary server commit to a random secret to be used later as an unbiased source of randomness. Concretely, when the client visits the publisher, the publisher contacts the auxiliary server, notifies it of an incoming auction, and supplies to it a commitment to a uniform random secret, $secret_p$. The auxiliary server replies with its own commitment to a uniform random secret, $secret_a$. They keep these secrets hidden until Step 4.

Step 2: Encode and split bids. The browser sends an invitation to selected bidders with the auction's id. If a bidder wishes to participate, they encode their bid using the Encode-MAX procedure (§4.2), split the resulting vector into two additive shares as discussed in Section 4.3, and generate a fresh signing and verification key pair. The verification key acts as the bidder's *bidder id* in the auction. The bidder then sends share M^1 to the auxiliary server and share M^2 to the publisher, supplying both with the bidder id. Bidders who fail to submit their shares before a timeout are kept out of the auction.

Step 3: Find the highest bid. Before computing the auction, the auxiliary server sends to the publisher all the bidder ids that it received, the publisher matches them with the ids that it received, and responds to the auxiliary server with the intersection. The publisher and the auxiliary server then use the vector shares of bidders in the intersection in the MAX protocol of Section 4.3. If Decode-MAX produces an invalid unary vector such as [1,0,1,0], the auction is aborted; when parties are honest, abort happens with negligible probability (§4.2). At the end, both parties learn the highest bid, b^*, but nothing else. To avoid parties adapting their sum vector share in response to the other's sum vector share, parties first exchange commitments of their sum vector share; the honest party aborts if misbehavior is detected.

Lemma 1. Let $b_1, \ldots, b_j$ be the bids from j honest bidders, and let $M_1, \ldots, M_j$ be the AFE vectors resulting from running Encode-MAX on the bids. Similarly, let $M_{j+1}, \ldots, M_{j+k}$ be AFE vectors that MAX-encode bids from k malicious bidders (these AFE vectors can represent invalid unary vectors like [1,0,1,0]). Decode-MAX on the sum of these $j+k$ AFE vectors outputs, with high probability, either an invalid value (invalid unary representation) or a value $\geq max(b_1, \ldots, b_j)$.

Lemma 2. Let M^1 and M^2 be sum vector shares held by the auxiliary server and publisher, respectively. During Decode-MAX, if the auxiliary server uses a different sum vector share M'^1 without having seen M^2 first or the publisher uses a different sum vector share M'^2 without having seen M^1 first, then the output of Decode-MAX is, with high probability, either an invalid value (invalid unary representation) or ℓ.

Appendix A gives proofs for both lemmas. Together they imply that malicious actions by participants lead to the resulting highest bid being invalid or at worst larger than the real highest bid. Either outcome leaks no information about honest losing bidders' bids to the attacker (our privacy goal). Furthermore, malicious actions are detected by Addax's verification.

Step 4: Find the winner. The publisher and the auxiliary server find the *winner* (the bidder id of the party who submitted b^*) interactively. First, both parties decommit to the secrets they generated in Step 1, check the decommitment, and XOR the secrets together to obtain $secret = secret_a \oplus secret_p$. Since at least one party is honest, *secret* is uniformly random and independent of the bidder ids generated by the bidders; the parties use *secret* as the seed to a pseudorandom generator (PRG). Both parties locally use the PRG to pick the same random bidder w from the set of participating bidder ids, which avoids biasing the auction towards a particular bidder in the case of ties (the PRG is for fairness not for privacy). The auxiliary server sends the b^*-th value of its share of bidder w's vector, $M_w^1[b^*]$, to the publisher and the publisher sends $M_w^2[b^*]$. Both parties then locally sum the two shares to obtain $M_w[b^*] = M_w^1[b^*] + M_w^2[b^*]$. Applying Decode-OR to $M_w[b^*]$ yields β_{b^*}, which is the bit of bidder w at position b^* in the unary vector (§4.2). If β_{b^*} is 0, bidder w is not the winner (since its bid must be lower than b^*). Note that learning β_{b^*} reveals no additional information. The publisher and the auxiliary server continue to pick a random bidder id w until the bit β_{b^*} of w is 1 ($n/2$ tries in expectation). In such a case, w is the winner. Finally, the auxiliary server and publisher ask w if its bid is b^*. Bidder w replies only if it receives the same query from both parties. If w's bid is not b^*, it sends *abort* to both parties and the auction is aborted. If there are ties (i.e., multiple bidders submitted b^*), this procedure returns a uniformly chosen one. The ids of other tied bidders remain hidden.

Lemma 3. In Step 4, if the auxiliary server sends to the publisher an AFE share that is different than what it received from the candidate winner w (i.e., different from $M_w^1[b^*]$) or the publisher sends to the auxiliary server an AFE share that is different than what it received from w, then the auction aborts or w is declared the winner with high probability. In the latter case, w is either the real winner or a malicious bidder.

Appendix A proves this lemma. It basically means that a malicious publisher or auxiliary server can only ever make a colluding bidder the winner; they cannot cause a winner (if chosen by the PRG) to lose, nor can they make an honest losing bidder the winner (and hence learn its bid).

Step 5: Compute the sale price. The above four steps are sufficient to compute first-price auctions (the most common type) where the winner is the highest bidder and the sale price is its bid. To support second-price auctions (sometimes used by exchanges), the auxiliary server and publisher subtract the winning bidder's vector share from the sum vector share (e.g., the auxiliary server subtracts M^1_w from M^1). They then rerun Step 3 to obtain the second highest bid.

Lemma 4. If either the auxiliary server or the publisher misbehaves in Step 5, or a malicious bidder is declared the winner in Step 4, then the computed sale price is, with high probability, either: (1) the highest bid among all bidders; (2) the second highest bid among all bidders; or (3) ℓ.

Appendix A has the proof of this lemma, which again hides the losing bids (besides the second-highest). Furthermore, any misbehavior is eventually detected during an audit.

5 Adding public verifiability

For an auditor to verify the outcome of an auction, the auditor needs to check that (1) the highest bid b^* selected in Step 3 of the auction is correct; (2) that the bit β_{b^*} of the winning bidder is 1 in Step 4; and (3) that the value computed in Step 5 was set as the auction's sale price. We start by making the output of AFEs publicly verifiable, and then discuss how an auditor can perform the above checks.

5.1 Verifiable and private AFEs

We make AFEs verifiable with a procedure that takes the result of the AFE computation—the sum vector $\mathbf{v}$—and commitments to the inputs, and outputs whether $\mathbf{v}$ is correct.

The key idea of our verification procedure is to observe that by their very nature, AFEs encode inputs in such a way that the desired functions (OR, MAX, etc.) can be computed with only additions. Hence, if one uses an *additively homomorphic commitment* scheme on the input AFE values, it is possible to check the result of the AFE computation without learning the inputs by adding the commitments and confirming whether the result is also a valid commitment of the output. We explain this process for the "OR" AFE of Section 4.1.

Encode V-OR. Given a bit $x \in \{0,1\}$, its verifiable AFE is a tuple v consisting of 2 elements in $\mathbb{Z}_p$ defined as follows. The first element in v is given by Encode-OR (§4.1). The second element in v is a non-zero uniform random element in $\mathbb{Z}_p$.

Commit V-AFE. Given a V-AFE tuple $v \in \mathbb{Z}_p^2$ encoding bit x with Encode V-OR, we use the Pedersen commitment [74] defined over a multiplicative group $\mathbb{G}$ of prime order p with generators $\{g,h\}$.[1] The commitment is $c = g^{v[0]} \cdot h^{v[1]}$.

This commitment *perfectly hides* the V-AFE tuple (an adversary cannot learn the tuple from the commitment); it *binds* the tuple (a committer cannot claim to have committed to a different tuple) if the discrete log problem is hard in $\mathbb{G}$. It is also additively homomorphic: given a commitment $c_1 \in \mathbb{G}$ to a tuple $v_1 \in \mathbb{Z}_p^2$ and a commitment $c_2 \in \mathbb{G}$ to a tuple $v_2 \in \mathbb{Z}_p^2$, $c_3 = c_1 \cdot c_2$ is a valid commitment to the tuple $v_1 + v_2$.

Compute and Decode. Given a set of n V-AFE tuples $\{v_1,\ldots,v_n\}$, which encode n bits $\{x_1,\ldots,x_n\}$ with the above Encode V-OR procedure, compute the OR of the n bits by adding the V-AFE tuples component-wise: $\mathbf{v} = v_1 + \cdots + v_n$. Decode V-OR calls Decode-OR on the first element in $\mathbf{v}$.

Verify V-OR. Given the V-AFE sum tuple $\mathbf{v}$ which encodes the result of the Compute V-OR procedure over n V-AFE tuples $\{v_1,\ldots,v_n\}$, and given a set of commitments $\{c_1,\ldots,c_n\}$ to these tuples generated with the Commit V-AFE procedure, one can verify $\mathbf{v}$ by checking if $g^{\mathbf{v[0]}} \cdot h^{\mathbf{v[1]}} \overset{?}{=} \prod_{j=1}^{n} c_j$. Verify V-OR outputs "ok" if the check passes, and "fail" otherwise.

The above approach generalizes to other functions (e.g., MAX) that require more complex encodings (e.g., vectors) since those encodings are just sets of AFE values. For example, a V-AFE vector is simply a vector of V-AFE tuples, and the commitment is a vector of Pedersen commitments—one for each tuple in the V-AFE vector. The approach can also be combined with secret sharing (§4.3) to hide the inputs from non-colluding parties. Specifically, the input providers (e.g., bidders in our case) generate their V-AFE vectors $\{M_1,\ldots,M_n\}$ and compute the corresponding commitments $\{c_1,\ldots,c_n\}$, which are made available on a public log. Then, the input providers generate secret shares for their V-AFE vectors and give these shares to the computing parties as described in Section 4.3. Finally, the computing parties combine their sum vector shares into the V-AFE vector $\mathbf{M}$ and verify each entry with Verify V-OR and the commitments.

5.2 Verifiable, private, and decentralized auction

We now discuss how to extend the protocol of Section 4.4 with the V-AFE construction of Section 5.1 to obtain verifiability of the auction's outcome in addition to privacy.

Recall that in Step 2 of the auction protocol (§4.4), a bidder i encodes its bid using Encode-MAX (§4.2) which produces an AFE vector M_i, where each entry in M_i is an Encode-OR (§4.1) of each bit of bidder i's unary-formatted bid. In our verifiable auction, the bidder instead uses the Encode V-OR procedure (§5.1), so M_i is made up of ℓ V-AFE tuples. Bidder i also creates, for each entry of M_i, a commitment using Commit

[1] As an (insecure) example, the set $\{1,3,4,5,9\}$ in $\mathbb{Z}_{11}$ forms a multiplicative group with 5 elements (its order is $p = 5$). A generator for this group is 3 since repeated multiplications of 3 with itself generates every element.

V-AFE (§5.1). Let C_i denote the corresponding vector of ℓ commitments for M_i. Bidder i then splits M_i (§4.3), and sends to the auctioneers a collision-resistant hash of C_i and the AFE vector shares (M_i^1 or M_i^2, depending on the party).

Asynchronously, bidder i uploads to the public log (§7) its bidder id, C_i, and a signature of C_i that validates with the bidder id (recall that bidder ids are verification keys). The other steps of the auction proceed as before. At the end of the auction, the publisher and the auxiliary server upload an *audit trail* to the public log containing: (1) the auction's outcome, consisting of the bidder id of the winner w, the highest bid b^*, and the auction's sale price; (2) their share of the sum vector computed in Step 3 and 5 of the auction protocol; (3) the b^*-th entry of the V-AFE vector share of each candidate winner chosen in Step 4 and the seed for the PRG used; and (4) the hashes (to commitments) they received from bidders.

Deferred public verification. After the auction completes, an *auditor* can choose to verify that the auction was done correctly as follows. The auditor accesses the auction's audit trail from the public log, and verifies that the uploaded hashes match the commitments, and all signatures on the commitments are valid. To verify the highest bid in Step 3, the auditor aggregates the sum vector shares in the audit trail to obtain $\mathbf{M}$. Then, the auditor computes the highest bid b^* by calling Decode-MAX on $\mathbf{M}$ (§4.2). Finally, the auditor runs, for all $j \in [1, \ell]$, Verify V-OR (§5.1) using as input the j-th entry of $\mathbf{M}$ (acting as the V-AFE sum value), and the j-th entry of every commitment vector submitted by the n bidders (i.e., for all $i \in [1, n]$, $C_i[j]$), as the commitment set. If all checks pass, then Step 3 was correct. The auditor performs the same actions for Step 5 to verify the second highest bid.

To verify Step 4, the auditor checks, for each of the candidate winners x, whether $g^{M_x[b^*][0]} \cdot h^{M_x[b^*][1]} \overset{?}{=} C_x[b^*]$. The auditor also checks that the Decode-OR of $M_w[b^*]$ is 1 (i.e., the actual winner's bit at position b^* is indeed a 1), and the Decode-OR of $M_x[b^*]$ for all other candidate winners x is 0. Then, the auditor uses the PRG and the seed in the audit trail to check that the bidder ids of the set of candidate winners are correct and that w was the last bidder id sampled.

Theorem 1. Addax's auction protocol with deferred verification achieves completeness, soundness, and privacy.

We give the full definitions and proofs in Appendix B. Note that detection is different from finding the party at fault.

5.3 Assigning blame

An auction may be aborted during the online phase, or deferred verification may fail. In these cases, Addax can narrow down the set of faulty parties. As parties participate in many auctions (recall that exchanges process billions of auctions per day), one could develop detection algorithms that flag those who are present in an unusually high number of aborted or failed auctions. We discuss this in more detail in Appendix D.

6 Optimizations

This section discusses two optimizations. The first adds interaction between the bidders and the auctioneers to dramatically cut costs. The second reduces interaction between the auctioneers, which lowers latency, but leaks the existence of ties.

6.1 Less communication with an interactive MAX

A major drawback of the proposed private auction protocol is that the computation and communication complexity of computing MAX using AFE vectors and their corresponding shares is $O(\ell)$, where ℓ is the highest possible bid (§4.2). Meanwhile, bids range from cents to tens of dollars; a realistic deployment would need $\ell \geq 1{,}000$, which is too costly. In this section we show how to modify the auction protocol to add r rounds of interaction between bidders and the *auctioneers* (publisher and auxiliary server) in exchange for reducing computation and communication complexity to $O(r \cdot \ell^{1/r})$.

High-level idea. In Figure 2, bidders first represent their bids as a unary bit vector, and then use Encode-OR on each bit to create vector M. This vector is then split into shares M^1 and M^2. The auctioneers aggregate their shares locally and then exchange their sum vector shares to construct the sum vector $\mathbf{M}$. This vector is then decoded into a unary bit vector that contains the result of max. Observe that if the bidders were to use Encode-OR only on the last two bits of their bit vectors (the gray and light gray cells), they would obtain the last 2 entries of M, which would then be split into the last two entries of M^1 and M^2, and would become the last 2 entries of the sum vector shares, and finally of $\mathbf{M}$. Decoding these two entries of $\mathbf{M}$ results in the last two bits of the final unary bit vector (in the example these bits are 1 and 0). The fact that the last bit is 0 means that the max value must be $< \ell$. The fact that the penultimate bit is 1 means that the max value must be $\geq \ell - 1$. Hence, encoding and sharing only a subset of bidders' unary bit vectors is enough to compute the max value. Of course, in this example we knew ahead of time which two elements to pick to get a tight upper and lower bound on the max. In our protocol, the auctioneers do r rounds of k-ary search ($k = \ell^{1/r}$) to find the consecutive positions at which the final unary bit vector changes from a 1 to a 0, which yields the max.

Protocol. Using the notation of Section 4.3, each bidder i sends $\lceil \ell^{1/r} \rceil$ entries of the AFE vector shares M^1 and M^2 to the auctioneers in each round. The entries sent in each round are evenly distributed between the current lower and upper bounds on the maximum bid (initially set to 1 and ℓ, respectively). For each of the chosen entries j, the auxiliary server runs the Compute-OR procedure (§4.1) by aggregating the shares it receives from each bidder i: $M^1[j] = \sum_i M_i^1[j]$. Likewise, the publisher computes $M^2[j] = \sum_i M_i^2[j]$. The publisher and the auxiliary server then exchange their sum shares for each entry j, allowing the reconstruction of $\mathbf{M}[j] = M^1[j] + M^2[j]$. Calling Decode-OR (§4.1) on $\mathbf{M}[j]$ returns whether bit β_j in the unary vector is 1 or 0. If β_j is 1, the highest bid $b^* \geq j$. Else, $b^* < j$.

This establishes a new lower and upper bound on b^* with respect to the exchanged entries. After r rounds, the number of entries sent by each bidder to each auctioneer is $\leq r \cdot \lceil \ell^{1/r} \rceil$.

In this protocol, bidders transmit a subset of the entries that they send to the auctioneers in the non-interactive variant, and hence they reveal less information. But there is one downside: bidders or an auctioneer can adaptively send inconsistent shares in response to partial information (e.g., knowledge that the max is in a given range). This could affect the auction's integrity. Addressing this issue requires extending the protocol with two extra safeguards: (1) an asynchronous step to find the sale price bidder which is similar to Step 4 in Section 4.4; and (2) generating a zero-knowledge proof that the sale price bidder's AFE vectors are valid without leaking the original AFE vector. Appendix C describes these steps in detail and proves the following two lemmas.

Lemma 5. If either the auxiliary server or the publisher misbehaves, or malicious bidders issue inconsistent AFE shares, the above interactive protocol leaks no more information about losing bidders' bids than the non-interactive variant.

Lemma 6. If either the auxiliary server or the publisher misbehaves, or malicious bidders issue inconsistent AFE shares, the above protocol (with the extra safeguards) ensures that malicious actions are detected during an audit.

6.2 Lower latency by leaking the existence of ties

Of all the steps in the auction protocol, finding the winner (§4.4, Step 4) is the most expensive since each interaction between the publisher and the auxiliary server occurs over WAN. This step consists of two parts: (1) pick a random candidate winner w, and (2) exchange the b^*-th entry of w's AFE vector shares to determine whether w indeed had the highest bid—trying again otherwise. The iterated nature of this algorithm aims to find one of the highest bidders at random (as soon as a highest bidder w is found, the auctioneers halt). One can eliminate this cost if one is willing to leak the number of ties. The protocol is simple: the publisher and auxiliary server exchange the b^*-th entry of the vector shares of *all* bidders. In the absence of ties, after decoding, only one bidder will have a 1 and all others will have a 0. If there are ties, multiple bidders will have a 1 at position b^*, and the auctioneers use the PRG to break the tie. Addax adopts this tradeoff.

We note that the added leakage is actually minor given that in the interactive protocol (Step 4), one learns that it took k tries to find the winner. In an auction with no ties, k would be $n/2$ in expectation, so the value of k already leaks some information about the number of potential ties that may exist.

7 Search and filtering

In Addax, bidders register to participate in auctions by storing their information (e.g., ad categories, domain of their bidding service) on a public tamper-proof log, and auction participants also use this log to create an audit trail (§5.2). Our implementation uses the Algorand blockchain [3] to maintain the log, though we could have used a BFT consortium or a trusted party (if one exists). Addax also needs a way to *search* the blockchain. This is typically done by downloading the entire blockchain and locally searching for the desired objects. Of course, this is onerous for browsers, as no user would ever maintain a copy of the blockchain just to receive ads. Instead, our implementation uses the Purestake indexer [14]. The downside is that one must trust this indexer. One way to remove this assumption is to use a verifiable search engine for blockchains [69].

Even with the Purestake indexer, querying data is slow: it takes seconds to get a response. Therefore, Addax keeps a copy of the log in *untrusted* cache servers; Addax then queries Purestake asynchronously to verify the cache servers' results. Querying cache servers takes only a few milliseconds.

In the rest of this section we describe how browsers do local filtering and fetch advertisers' data. We discuss how browsers interact with the Algorand blockchain in Appendix E.

7.1 Filtering and inviting advertisers

Upon visiting a page with ads and obtaining a list of allowed categories from the publisher, the browser queries the cache server to get bidders who match these categories. The browser caches bidder information and only sends "if-modified-since" requests to the cache server to reduce communication. Borrowing ideas from Privad [58] and Adnostic [80], the browser assigns a preference score for each of the returned bidders. The browser then picks the top k bidders and invites them to join the auction, supplying them with information about the publisher and the user. Depending on the configuration of Addax, the user information can be empty (for generic ads), include a group or topic id (as in FLoC [89] and Topics [16]), or include cookies and demographic information. Since the publisher's revenue depends on bids, and bidder valuations are based on user information, different publishers can require different levels of information disclosure to access their content. This is similar to how publishers detect ad blocking software and request that users disable it.

8 Implementation

Addax consists of 2.2K lines of C++ and 400 lines of Python and PyTeal [15] for Algorand smart contracts. Addax's client-side tracking is done outside the browser and interacts with Chrome via native messaging [9]. We use OpenSSL 3.0.0 [12] for basic cryptographic operations (e.g., `BN_rand` as the PRG). Addax's Pedersen commitment (§5.1) is defined over elliptic curve `secp192r1`, as is the Schnorr signature scheme [78] that bidders use to sign their log entries. Elements in V-AFE vectors are defined over the 192-bit field used in `secp192r1`.

Baselines. To contextualize our contributions, we implement baselines using state-of-the-art homomorphic encryption (HE) and secure two-party computation (2PC) frameworks:

- *CKKS* on *SEAL* [25, 47]: HE for arithmetic operations.
- *TFHE* [49, 50]: HE for boolean operations.
- *MASCOT* on *MP-SPDZ* [65, 66]: Arithmetic 2PC.
- *ag2pc* on *EMP toolkit* [85, 86]: Boolean 2PC.

Homomorphic encryption. The publisher generates cryptographic keys and sends the public key to bidders. Bidders send bids encrypted with the public key to the auxiliary server, who runs the auction over ciphertexts and supplies the result to the publisher for decryption with the secret key. For SEAL we implement and measure the `maxId` algorithm by Cheon et al. [48] which is the best known way to find the ciphertext with the max value. While this is a subset of running an auction, this one step is already more expensive than Addax's full auction protocol. For TFHE we implement the whole auction. Neither baseline provides integrity.

Multiparty computation. Advertisers commit to their bids and send them to the publisher and auxiliary server alongside additive shares of their bids and the commitment randomness. Inside the MPC, the auxiliary server and publisher reconstruct the bids and the commitment randomness from their shares, check that the commitments match, and the bids are the committed values, and then run the auction using the bids. For commitments we use $H(rand||bid)$ and assume H is a random oracle. We use hash functions already implemented and optimized for these frameworks (e.g., SHA3, SHA256, MiMC).

9 Evaluation

This section studies the following questions:

1. What are the costs of Addax's auction for each party?
2. How does Addax's auction compare with alternatives?
3. What is the resource overhead of deploying Addax over a non-private and unverifiable exchange?
4. How expensive is the verification procedure?

Appendix E.3 discusses the cost of interacting with the log.

Evaluation environment. We run our experiments across AWS data centers to account for Addax's decentralized nature. The publisher is in US East (Ohio) on a c5.2xlarge instance, the auxiliary server in US West (OR) on a c5.2xlarge instance, and bidders in US West (CA) on c5.12xlarge instances. We use standard Ubuntu 20.04 for all of them. PureStake exposes a REST API and runs on servers in Ontario, CA, and OR.

Method and metrics. Our key metrics are the end-to-end latency, total network communication, and throughput of the auction procedure. This includes the events after the browser fetches the page from the publisher and initiates the auction, but before the browser fetches and displays the ad on the user's screen. In short, we measure the overhead of Addax over the status quo of using a centralized non-private ad exchange. We report the mean over 20 trials and one standard deviation. *We focus on second-price auctions in this evaluation*, as they are the more complex type of auction. If Addax is used for first-price auctions, the costs are 30% lower: auctions with 96 bidders complete within 440 ms, and Addax can sustain a throughput of 360 auctions per second per core.

	Size (MB)		Generation (ms)
AFE vector shares	0.48		87.55
Materials (non-interactive)	0.25		537.9
Materials (interactive)	1.705		1,802.0
	Non-interactive	**2-round**	**4-round**
Communication (MB)	0.48	0.0144	0.0034

FIGURE 3—Size of AFE vector shares and other materials (e.g., commitments), their generation time, and the total communication between a bidder and one auctioneer under different Addax variants.

Parameters. Prior reports [92] suggest that the typical number of bidders (usually demand-side platforms) in an auction is under 30. We experiment with up to 96 bidders, but Addax could handle more with little extra latency since most of the latency comes from round trips between the two servers and is not impacted by the number of bidders. We set $\ell = 10{,}000$, which supports bid ranges consistent with those observed in practice [93]. This results in a probability of computing the wrong MAX of $\approx 1-(1-\frac{1}{2^{192}})^{10{,}000}$, which is negligible.

Our baseline implementations are generous: we use 13-bits for bids (4/5 of our bid range) and do not measure the time to receive shares or ciphertexts from bidders for any of them.

9.1 Microbenchmarks: Addax's auction protocol

To answer our first research question we microbenchmark the operations of each of the auction participants.

Bidder's cost. Before the auction starts, bidders encode their bids, commit to the encodings, and send their shares to the auctioneers. Figure 3 depicts the time required to generate an AFE vector, and the verification materials in both the non-interactive protocol (§5) and the interactive variant (§6.1) using 8 CPU threads. For the latter we include the cost of the safeguards detailed in Appendix C.2. As shown in the figure, generating these materials is more expensive than the time budgeted for an auction. However, AFE vectors are made up of random elements; the only dependence on bids is whether to use a uniform element or a zero (§4.1). As a result, all materials can be precomputed and kept aside. Furthermore, their generation is parallelizable: we get a 5.83× speedup with 6× more cores. We expect bidders to be able to maintain their desired throughput, albeit at a higher cost ($) than they incur today.

When the auction starts and the bidder decides on its bid, it can draw from the set of pre-generated materials to construct bid-specific AFE vector shares, commitments, and proofs. With pre-generated materials, bidders respond in 10 ms.

Local auction computation. To determine the costs to the auxiliary server and the publisher we run a microbenchmark where both auctioneers run on the same machine, are given all materials (e.g., AFE shares), and compute the auction without the effects of network latency. Figure 4a shows the time for

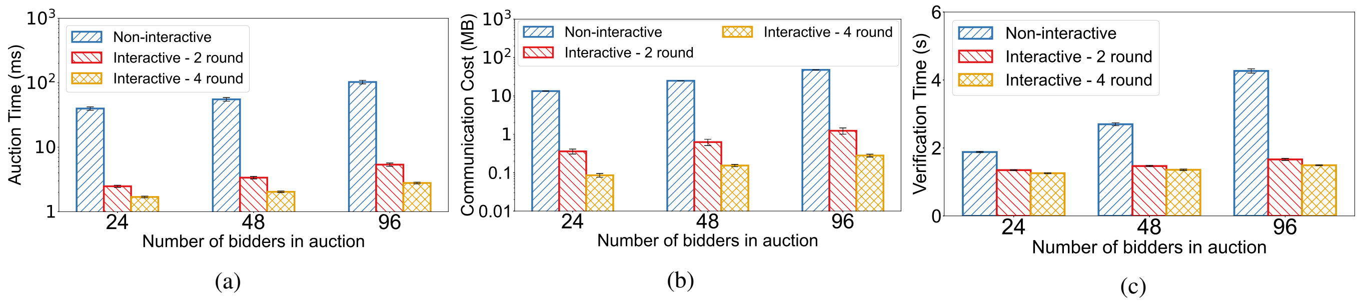


FIGURE 4—Cost of running and verifying an auction across Addax's variants. Figure (a) depicts the processing time incurred by each auctioneer; (b) depicts the communication costs for each auctioneer; and (c) depicts the costs to an auditor.

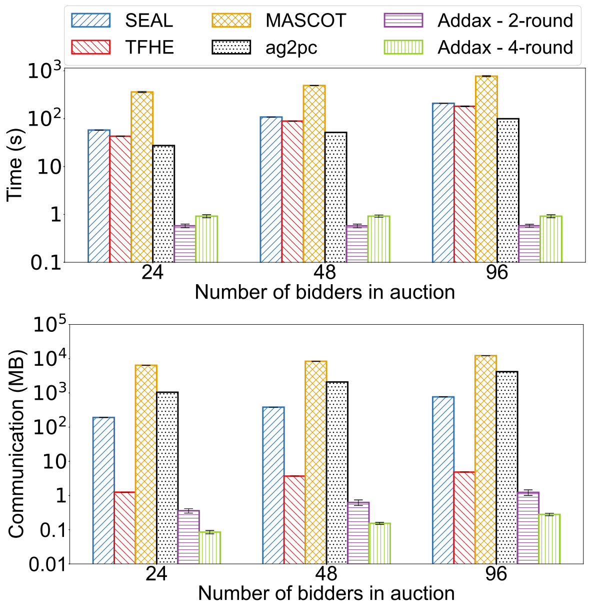


FIGURE 5—End-to-end latency and communication costs for an auction in Addax and several baselines over WAN.

different Addax variants. Compared to the non-interactive protocol, Addax with 4 rounds (§6.1) requires fewer operations since it acts on a tiny subset of the entries of the AFE vector, and reduces computation time for a 96 bidder auction from 102 ms to 2.8 ms. Interactivity also reduces communication costs for effectively the same reason (acts on fewer entries). As we show in Figure 4b, the size of the AFE shares exchanged between bidders and each auctioneer in the non-interactive variant of Addax with 96 bidders is 47.68 MB, whereas it is 0.28 MB with 4-rounds and 1.23 MB with 2-rounds.

9.2 End-to-end performance

The above microbenchmarks give an idea of the computation and communication costs that are expected when running auctions with Addax. However, the metric that actually matters is end-to-end latency over WAN. Figure 5 shows the computation and communication costs of Addax's end-to-end protocol over a WAN deployment, from the time that the publisher starts the auction, to the time the winner is notified. This figure

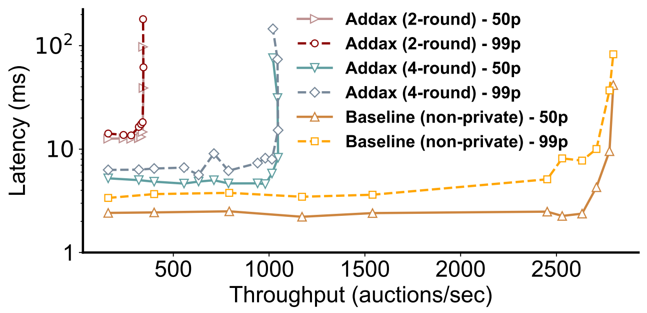


FIGURE 6—Median and 99-percentile response time and server throughput for Addax and a non-private baseline for an auction with 96 bidders. Each data point represents the latency and the throughput achieved at a given load (low and to the right is better).

also shows the baselines described in Section 8.

In terms of auction latency, Addax's 2-round variant is by far the most efficient option, often by orders of magnitude compared to the baselines. Addax's 2-round variant beats the 4-round variant due to fewer WAN RTTs at a slight increase in the amount of communication. At 96 bidders, the browser receives an ad tag from Addax in 579 ms; behind the scenes, the auctioneers exchange 1.23 MB of data to compute the auction. Of this time, the servers only spend 5 ms computing; the rest is network latency. Thus, having more bidders will not meaningfully increase the end-to-end latency of Addax.

For comparison, studies [1, 7, 20, 87] show that page loading times today take several *seconds*, so we expect Addax to run auctions asynchronously as the page loads without significantly impacting the user experience.

9.3 Costs over a non-private unverifiable baseline

To understand the additional computational resources required to deploy Addax, we compare its throughput on a c5.2xlarge instance (4-core VM) to a baseline that simply finds the highest and second highest bids (the only non-trivial computation is establishing a TLS session between the browser and the publisher). We run an open-loop workload with varying load and with all inputs already in-memory, so we do not measure network latency. Figure 6 gives the results.

Addax's 2-round and 4-round variant achieve 8.1× and

	Non-interactive	2-round	4-round
Auctioneers	1.932	0.056	0.025
Bidders	0.250	0.250	0.250
Winner	0.250	0.730	0.730
Sale price bidder	0.250	1.705	1.705

FIGURE 7—Total size (MB) of the audit information that parties must upload to the public log in an auction of 96 bidders.

2.7× lower throughput than the baseline. As Addax requires two servers, this translates to 16.2× and 5.4× more computation resources to maintain the same throughput as the baseline. This suggests that Addax could process the high volume of auctions that exchanges process today while providing integrity and privacy guarantees—albeit at a premium cost.

9.4 Cost of verification

In this section we evaluate the cost for auction participants to supply the necessary materials to leave an audit trail, and for an auditor to validate the correctness of an auction.

Leaving an audit trail. After the auction finishes, participants upload their audit materials (§5.2–§5.3) to Algorand. This takes around 0.8 sec. Figure 7 gives the size of the materials that each party uploads for a 96-bidder auction. In the interactive variants, the winner uploads its full AFE vector, and the sale price bidder uploads its full AFE vector with a random mask and proofs as described in Appendix C.5.

Verification time. Verification requires downloading the materials from Algorand, checking the hash of commitments, and checking the recovered AFE sum vectors and bit encodings (§5.2). Auditors also need to validate that AFE vectors from the winner and sale price bidder are valid (§C.2). Figure 4c depicts the time of verification. In the non-interactive variant, deserializing commitments and verifying the two sum vectors takes most of the time. In contrast, in the interactive variants the expensive step is validating the winner and sale price bidders' AFE vectors. To verify a 96-bidder auction, the non-interactive variant requires 4.27 sec, while the 2-round and 4-round variants take 1.66 sec and 1.49 sec, respectively.

10 Related work

This section describes other efforts that relate to Addax.

Advertising. There is a rich literature in privacy preserving ads [35, 44, 52, 57–59, 75, 76, 80], but none focuses on private and verifiable auctions. VEX [35] provides verifiability but the auctioneer learns all bids. Privad [58, 59], Adnostic [80], FloC [89], Topics [16], and others [37, 57, 76] reduce the collection of user information, but auctions are still conducted by a party that learns all bids and cannot be audited.

Private and verifiable auctions. In other domains, there is work on private or verifiable auctions. Parkes et al. [73] provide auction integrity but the auctioneer learns all bids, unlike Addax. Other works [62, 67] provide privacy but not integrity. Finally, there are several multiparty protocols [38, 42] where the bidders jointly compute the auction. This is worse than our MPC baseline in Section 8 in that here bidders actively participate in the protocol rather than merely generating shares. This does not scale to more than a handful of bidders.

11 Discussion

Addax departs from the status quo by introducing accountability to an opaque ecosystem. While this is a disruptive change, there are two things on Addax's favor. First, the ad-tech industry already uses browsers to kickstart auctions and invite bidders [4] and newer proposals like Google's FLEDGE [33] push even more functionality to browsers include client-side tracking. Second, Addax is incrementally deployable: an Addax-enabled browser can send an HTTP X-header indicating its support of the protocol, and interested publishers can respond with Addax-based ad spots while continuing to offer traditional ads to other users. Furthermore, we think many missing features can be implemented in Addax.

Content curation. A key role of exchanges is to prevent *malvertising* (the use of ads to spread malware) or ads that can damage the publisher's brand. On the one hand, content curation is hard even in centralized environments: reports of malicious actors leveraging ad networks to distribute malware are common [8]. On the other hand, since advertisers publish their information on Addax's public log, one could imagine requiring advertisers to upload their ads as well. Then, just like existing services scan blockchains for anomalous transactions, they can scan Addax's log to detect and flag malicious ads.

Fraud prevention. Many existing mechanisms to prevent *publisher fraud* (e.g., using clickbots to increase revenue) [79] still work in our setting. For example, bidders can still observe anomalous changes in ad traffic from a publisher, and can perform randomized auditing with *bluff ads* [60] (uninviting ads unlikely to be clicked by real users). Other techniques that collect hard-to-fake signals from a device with the aim of detecting bots [18, 21] could also be used, but more work is needed to port them to our context.

Conversions. Analytics are also critical to the ad ecosystem. Currently, advertisers and publishers rely on third-party cookies to track when a user performs an action after viewing an ad (a "conversion"). A recent proposal [94] shows how this can be done without cookies and without learning the user's identity; this approach is compatible with Addax's architecture.

Trust-performance tradeoff. Our description of Addax uses 2 parties but the protocols naturally generalize to k auxiliary servers; if either the publisher or any of the k auxiliary servers is honest, Addax provides its guarantees. Of course, as the number of parties increases the costs also increase. This tradeoff can be taken into account at deployment time.

Acknowledgments

We thank the OSDI and NSDI reviewers, and our shepherd, Bryan Ford, for their thoughtful comments that improved this paper. This work was funded in part by NSF grant CNS-2045861 and DARPA contract HR0011-17-C0047.

References

[1] About pagespeed insights. https://developers.google.com/speed/docs/insights/v5/about.

[2] Adtelligent's header bidding platform. https://adtelligent.com/header-bidding-platform/.

[3] Algorand. https://www.algorand.com/.

[4] The beginner's guide to header bidding. https://adprofs.co/beginners-guide-to-header-bidding/.

[5] Google Has a New Plan to Kill Cookies. People Are Still Mad. https://www.wired.co.uk/article/google-floc-cookies-chrome-topics.

[6] Google's Topics API: Rebranding FLoC Without Addressing Key Privacy Issues. https://brave.com/web-standards-at-brave/7-googles-topics-api/.

[7] Here's what we learned about page speed. https://backlinko.com/page-speed-stats.

[8] Malvertising: What is it and how to avoid it. https://us.norton.com/internetsecurity-malware-malvertising.html.

[9] Native messaging. https://developer.chrome.com/docs/apps/nativeMessaging/.

[10] Openrtb protocol buffer 2.5.0. https://developers.google.com/authorized-buyers/rtb/downloads/openrtb-proto.

[11] OpenRTB (real time bidding). https://www.iab.com/guidelines/real-time-bidding-rtb-project/.

[12] OpenSSL. https://www.openssl.org.

[13] Parakeet. https://github.com/WICG/privacy-preserving-ads/blob/main/Parakeet.md.

[14] Purestake. https://www.purestake.com/.

[15] Pyteal: Algorand smart contracts in python. https://github.com/algorand/pyteal.

[16] The Topics API. https://developer.chrome.com/docs/privacy-sandbox/topics/.

[17] This is how Google plans to track you now. https://www.slashgear.com/this-is-how-google-plans-to-track-you-now-25708910/.

[18] What is recaptcha? https://www.google.com/recaptcha/about/.

[19] Cookie synching. https://www.admonsters.com/cookie-synching/, 2010.

[20] Find out how you stack up to new industry benchmarks for mobile page speed. https://think.storage.googleapis.com/docs/mobile-page-speed-new-industry-benchmarks.pdf, 2017.

[21] Fighting fraud using partially blind signatures. https://engineering.fb.com/2019/10/16/security/partially-blind-signatures/, 2019.

[22] Iab internet advertising revenue report. https://www.iab.com/wp-content/uploads/2019/05/Full-Year-2018-IAB-Internet-Advertising-Revenue-Report.pdf, 2019.

[23] Cookie matching. https://developers.google.com/authorized-buyers/rtb/cookie-guide, 2020.

[24] Lattigo v2.1.1. Online: http://github.com/ldsec/lattigo, Dec. 2020.

[25] Microsoft SEAL (release 3.6). https://github.com/Microsoft/SEAL, Nov. 2020.

[26] PALISADE Lattice Cryptography Library (release 1.10.6). https://palisade-crypto.org/, Dec. 2020.

[27] Private marketplace ad spending to surpass open exchange in 2020. https://www.emarketer.com/content/private-marketplace-ad-spending-to-surpass-open-exchange-in-2020, 2020.

[28] Antitrust battle latest: Google, facebook 'colluded' to smash apple's privacy protections. https://www.theregister.com/2021/10/22/google_facebook_antitrust_complaint/, 2021.

[29] Azure attestation client library for .net - version 1.0.0. https://docs.microsoft.com/en-us/dotnet/api/overview/azure/security.attestation-readme, 2021.

[30] Bring more bids to the auction with open bidding. https://admanager.google.com/home/resources/feature-brief-open-bidding/, 2021.

[31] SCALE and MAMBA. https://github.com/KULeuven-COSIC/SCALE-MAMBA, 2021.

[32] The comprehensive guide to online advertising costs. https://www.wordstream.com/blog/ws/2017/07/05/online-advertising-costs, 2022.

[33] Fledge api. https://developer.chrome.com/docs/privacy-sandbox/fledge/, 2022.

[34] E. Anderson, M. Chase, F. B. Durak, E. Ghosh, K. Laine, and C. Weng. Aggregate measurement via oblivious shuffling, 2021. https://ia.cr/2021/1490.

[35] S. Angel and M. Walfish. Verifiable auctions for online ad exchanges. In *Proceedings of the ACM SIGCOMM Conference*, 2013.

[36] Y. Aumann and Y. Lindell. Security against covert adversaries: Efficient protocols for realistic adversaries. *Journal of Cryptology*, 23(2), 2010.

[37] M. Backes, A. Kate, M. Maffei, and K. Pecina. Obliviad: Provably secure and practical online behavioral advertising. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2012.

[38] S. Bag, F. Hao, S. F. Shahandashti, and I. G. Ray. Seal: Sealed-bid auction without auctioneers. *IEEE Transactions on Information Forensics and Security*, 15, 2020.

[39] M. A. Bashir and C. Wilson. Diffusion of User Tracking Data in the Online Advertising Ecosystem. In *Proceedings of the Privacy Enhancing Technologies Symposium (PETS)*, 2018.

[40] A. Bois, I. Cascudo, D. Fiore, and D. Kim. Flexible and efficient verifiable computation on encrypted data. Cryptology

ePrint Archive, Report 2020/1526, 2020.

[41] F. Bourse, O. Sanders, and J. Traoré. Improved secure integer comparison via homomorphic encryption. In *Topics in Cryptology – CT-RSA 2020*, 2020.

[42] F. Brandt. A verifiable, bidder-resolved auction protocol. In *Proceedings of the 5th International Workshop on Deception, Fraud and Trust in Agent Societies*, 2002.

[43] J. V. Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. Wenisch, Y. Yarom, and R. Strackx. Foreshadow: Extracting the keys to the intel sgx kingdom with transient out-of-order execution. In *Proceedings of the USENIX Security Symposium*, 2018.

[44] J. Cartlidge, N. P. Smart, and Y. Talibi Alaoui. Mpc joins the dark side. In *International Symposium on Information, Computer, and Communications Security*, 2019.

[45] F. Chanchary and S. Chiasson. User perceptions of sharing, advertising, and tracking. In *Symposium On Usable Privacy and Security (SOUPS)*, 2015.

[46] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai. Sgxpectre attacks: Leaking enclave secrets via speculative execution. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2019.

[47] J. H. Cheon, A. Kim, M. Kim, and Y. Song. Homomorphic encryption for arithmetic of approximate numbers. In *International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT)*, 2017.

[48] J. H. Cheon, D. Kim, D. Kim, H. H. Lee, and K. Lee. Numerical method for comparison on homomorphically encrypted numbers. In *International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT)*, 2019.

[49] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène. TFHE: Fast fully homomorphic encryption library, August 2016. https://tfhe.github.io/tfhe/.

[50] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène. TFHE: Fast fully homomorphic encryption over the torus. *Journal of Cryptology*, 33, 2020.

[51] H. Corrigan-Gibbs and D. Boneh. Prio: Private, robust, and scalable computation of aggregate statistics. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017.

[52] E. Deng, H. Zhang, P. Wu, F. Guo, Z. Liu, H. Zhu, and Z. Cao. Pri-rtb: Privacy-preserving real-time bidding for securing mobile advertisement in ubiquitous computing. In *Information Sciences*, 2019.

[53] A. Fiat and A. Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *Advances in Cryptology — CRYPTO' 86*, 1987.

[54] D. Fiore, A. Nitulescu, and D. Pointcheval. Boosting verifiable computation on encrypted data. In *Proceedings of the International Conference on Practice and Theory in Public Key Cryptography (PKC)*, 2020.

[55] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, and N. Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2017.

[56] A. Goldfarb and C. E. Tucker. Privacy regulation and online advertising. *Management Science*, 2010.

[57] M. Green, W. Ladd, and I. Miers. A protocol for privately reporting ad impressions at scale. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2016.

[58] S. Guha, B. Cheng, and P. Francis. Privad: Practical privacy in online advertising. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2011.

[59] S. Guha, A. Reznichenko, K. Tang, H. Haddadi, and P. Francis. Serving ads from localhost for performance, privacy, and profit. In *Proceedings of the ACM Workshop on Hot Topics in Networks (HotNets)*, 2009.

[60] H. Haddadi. Fighting online click-fraud using bluff ads. *ACM SIGCOMM Computer Communication Review*, 40(2), 2010.

[61] S. Halevi and V. Shoup. Design and implementation of helib: a homomorphic encryption library. Cryptology ePrint Archive, Report 2020/1481, 2020.

[62] M. Harkavy, J. D. Tygar, and H. Kikuchi. Electronic auctions with private bids. In *3rd USENIX Workshop on Electronic Commerce (EC 98)*, 1998.

[63] J. Horwitz and K. Hagey. Google's secret 'project bernanke' revealed in texas antitrust case. `https://www.wsj.com/articles/googles-secret-project-bernanke-revealed-in-texas-antitrust-case-11618097760`, Apr. 2021.

[64] I. Iliashenko and V. Zucca. Faster homomorphic comparison operations for BGV and BFV. Cryptology ePrint Archive, Report 2021/315, 2021.

[65] M. Keller. MP-SPDZ: A versatile framework for multi-party computation. Cryptology ePrint Archive, Report 2020/521, 2020.

[66] M. Keller, E. Orsini, and P. Scholl. Mascot: Faster malicious arithmetic secure computation with oblivious transfer. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2016.

[67] H. Kikuchi, S. Hotta, K. Abe, and S. Nakanishi. Distributed auction servers resolving winner and winning bid without revealing privacy of bids. In *Proceedings of the Seventh International Conference on Parallel and Distributed Systems: Workshops*, 2000.

[68] S. Lee, M. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In *Proceedings of the USENIX Security Symposium*, 2017.

[69] M. Li, T. Zhang, J. Zhu, C. Tan, Y. Xia, S. Angel, and H. Chen. Bringing decentralized search to decentralized services. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2021.

[70] Y. Lindell. How to simulate it - a tutorial on the simulation proof technique. Cryptology ePrint Archive, Report 2016/046, 2016. `https://ia.cr/2016/046`.

[71] J. R. Mayer and J. C. Mitchell. Third-party web tracking: Policy and technology. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2012.

[72] K. Murdock, D. Oswald, F. D. Garcia, J. Van Bulck, D. Gruss, and F. Piessens. Plundervolt: Software-based fault injection attacks against intel sgx. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (S&P'20)*, 2020.

[73] D. C. Parkes, M. O. Rabin, S. M. Shieber, and C. Thorpe. Practical secrecy-preserving, verifiably correct and trustworthy

auctions. *Electronic Commerce Research and Applications*, 2008.

[74] T. P. Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In *Proceedings of the International Cryptology Conference (CRYPTO)*, 1991.

[75] G. Pestana, I. Querejeta-Azurmendi, P. Papadopoulos, and B. Livshits. Themis: Decentralized and trustless ad platform with reporting integrity. `https://arxiv.org/abs/2007.05556v2`, 2020.

[76] A. Reznichenko, S. Guha, and P. Francis. Auctions in do-not-track compliant internet advertising. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2011.

[77] F. Roesner, T. Kohno, and D. Wetherall. Detecting and defending against third-party tracking on the web. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2012.

[78] C. P. Schnorr. Efficient identification and signatures for smart cards. In *Proceedings of the International Cryptology Conference (CRYPTO)*, 1989.

[79] B. Stone-Gross, R. Stevens, R. Kemmerer, C. Kruegel, G. Vigna, and A. Zarras. Understanding fraudulent activities in online ad exchanges. In *Proceedings of the ACM SIGCOMM Conference on Internet Measurement (IMC)*, 2011.

[80] V. Toubiana, A. Narayanan, D. Boneh, H. Nissenbaum, and S. Barocas. Adnostic: Privacy preserving targeted advertising. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2010.

[81] R. Tracy and J. Horwitz. Inside the Google-Facebook ad deal at the heart of a price-fixing lawsuit. `https://www.wsj.com/articles/inside-the-google-facebook-ad-deal-at-the-heart-of-a-price-fixing-lawsuit-11609254758`, Dec. 2020.

[82] J. Van Bulck, D. Moghimi, M. Schwarz, M. Lipp, M. Minkin, D. Genkin, Y. Yuval, B. Sunar, D. Gruss, and F. Piessens. LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2020.

[83] W. Vickrey. Counterspeculation, auctions, and competitive sealed tenders. *The Journal of Finance*, 1961.

[84] D. Wakabayashi and T. Hsu. Behind a secret deal between Google and Facebook. `https://www.nytimes.com/2021/01/17/technology/google-facebook-ad-deal-antitrust.html`, Jan. 2021.

[85] X. Wang, A. J. Malozemoff, and J. Katz. EMP-toolkit: Efficient MultiParty computation toolkit. `https://github.com/emp-toolkit`, 2016.

[86] X. Wang, S. Ranellucci, and J. Katz. Authenticated garbling and efficient maliciously secure two-party computation. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2017.

[87] X. S. Wang, A. Balasubramanian, A. Krishnamurthy, and D. Wetherall. Demystifying page load performance with wprof. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2013.

[88] D. I. Wolinsky, H. Corrigan-Gibbs, B. Ford, and A. Johnson. Scalable anonymous group communication in the anytrust model. In *Proceedings of the European Workshop on System Security (EUROSEC)*, Apr. 2012.

[89] Y. Xiao and J. Karlin. Federated learning of cohorts. `https://wicg.github.io/floc/`, 2021.

[90] Y. Xu, W. Cui, and M. Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2015.

[91] J. Yan, N. Liu, G. Wang, W. Zhang, Y. Jiang, and Z. Chen. How much can behavioral targeting help online advertising? In *International World Wide Web Conference (WWW)*, 2009.

[92] S. Yuan, J. Wang, B. Chen, P. Mason, and S. Seljan. An empirical study of reserve price optimisation in real-time bidding. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2014.

[93] W. Zhang, S. Yuan, J. Wang, and X. Shen. Real-time bidding benchmarking with iPinYou dataset. `https://arxiv.org/abs/1407.7073`, 2015.

[94] K. Zhong, Y. Ma, and S. Angel. Ibex: Privacy-preserving ad conversion tracking and bidding. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2022.

A Proofs for lemmas

Proof of Lemma 1. Let h be the maximum bid among all honest bidders. The event claimed in Lemma 1 is equivalent to one of two cases: (1) Decode-MAX outputs a valid unary bit vector whose value is larger than or equal to h; or (2) Decode-MAX outputs an invalid unary bit vector. Consider the opposite event where Decode-MAX outputs a valid unary bit vector whose value is smaller than h. We denote the probability of this event as $\Pr(\text{opposite})$. $\Pr(\text{opposite}) \leq \Pr(\text{Decode-OR outputs 0 at position } h) \leq 1/p$. Therefore, the probability that the output of Step 3 is greater than or equal to h or is an invalid unary bit vector is $1-\Pr(\text{opposite}) \geq 1-1/p$. In our construction p is a large prime, and hence $1-1/p \approx 1$.

Proof of Lemma 2. Let two parties hold additive shares for a given AFE value that was produced by Encode-OR. Let one of the parties be honest and the other malicious. Without seeing the share held by the honest party, the probability of the malicious party generating a share that results in Decode-OR outputting 0 is $1/p$: the malicious party would have to correctly guess the exact value needed to make the two shares add up to 0, and shares are uniformly random values in $\mathbb{Z}_p$. Thus, the probability of a malicious party generating AFE shares which lead to Decode-OR outputting 1 is $1-1/p$.

We use $\Pr(b)$ to denote the probability that Decode-MAX outputs b, and $\Pr(\text{invalid})$ to denote the probability that Decode-MAX outputs an invalid bit vector. Decode-MAX outputs b means that the decoded bit vector is $[\underbrace{1,\ldots,1}_{b},\underbrace{0,\ldots,0}_{\ell-b}]$.

Thus, $\Pr(b) = (1/p)^{\ell-b} \cdot (1-1/p)^{b}$.

$$\begin{aligned}
\Pr(0) + \ldots + \Pr(\ell - 1) &< (1/p)^{\ell} + \ldots + (1/p)^{1} \\
&= \frac{\frac{1}{p} - (\frac{1}{p})^{\ell+1}}{1 - \frac{1}{p}} \\
&< \frac{1}{p-1} \\
\Pr(\ell) + \Pr(\text{Invalid}) &= 1 - (\Pr(0) + \ldots + \Pr(\ell - 1)) \\
&\geq 1 - \frac{1}{p-1}
\end{aligned}$$

In our construction p is a large prime, so $1 - \frac{1}{p-1} \approx 1$.

Proof of Lemma 3. From Lemma 2 (and its proof), if a malicious party uses a bogus share for one of the bits, Decode-OR outputs 1 with probability $\geq 1 - 1/p$. Thus, if a malicious publisher or auxiliary server ever sends a bogus AFE share, Decode-OR would output 1 with high probability, leading to that candidate w becoming the winner. The auxiliary server and publisher then need to get acknowledgment from bidder w on whether its bid b_w is b^*. If w is honest and $b_w \neq b^*$, the auction aborts. If w is honest and $b_w = b^*$, w is the real winner, as b^* is greater than or equal to the real highest bid among all honest bidders with high probability. If w is malicious, it could abort or can choose to be the winner at its own discretion. If the latter, it must pay the second-highest bid (sale price).

Proof of Lemma 4. We consider two cases: (1) bidder w is honest and the real winner (highest bidder); (2) bidder w is a malicious bidder and not the highest bidder. We denote b as the computed sale price in Step 5.

For case (1), if the two auctioneers do not misbehave and use the correct inputs from bidders to compute Step 5, then b is the real sale price (i.e., the second highest bid) among all bidders. If a malicious auctioneer (auxiliary server or publisher) sends a sum vector share that is not computed correctly from bidders' inputs (i.e., the malicious auctioneer sends a sum vector share that is not $\sum_{i=1}^{N} M_i^1 - M_w^1$) in Step 5 when computing the sale price, b would be ℓ with high probability (Lemma 2).

For case (2), if the two auctioneers do not misbehave and use the correct inputs from bidders to compute Step 5, Step 5 finds the highest bid among all bidders excluding bidder w. Thus, b equals the highest bid among all bidders. If a malicious auctioneer sends an incorrect sum vector share (not computed correctly from the inputs of all bidders) in Step 5, then b is ℓ with high probability (Lemma 2).

B Proof for Addax's security properties

This section proves that Addax meets its security properties, which include auction completeness, soundness, and privacy.

Completeness. When all parties are honest, Addax's completeness relies on the probability of Decode-MAX being successful when it is used to find the highest bid and the sale price. Further, it relies on the probability of Decode-OR being successful when finding the id of the winner and the second highest bidder (recall this happens interactively by calling Decode-OR on a particular entry in the AFE sum vector of a candidate winner). In the worst case, Decode-OR might be run on up to n candidate winners ($2n - 1$ times for candidate winners and sale price bidders in the interactive variant). The probability of success is $\geq (1 - 1/p)^{2\ell} \cdot (1 - 1/p)^{n}$, and is $\geq (1 - 1/p)^{2\ell} \cdot (1 - 1/p)^{2n-1}$ in the interactive variant.

Soundness. There are three scenarios which result in an incorrect outcome: (1) publisher and auxiliary server are honest and some bidders are malicious; (2) either the publisher or the auxiliary server is malicious and all bidders are honest; (3) either the publisher or the auxiliary server is malicious and some bidders are malicious and colluding with the malicious auctioneer.

When bidders are malicious, they can: (A1) encode bids into an invalid unary bit vector (e.g., [1,0,0,1]), then generate AFE shares for such invalid unary bit vector and submit them to the publisher and auxiliary server; (A2) provide inconsistent commitments which are not commitments to the AFE it generates or provide inconsistent hash values of their commitments; (A3) claim to be the winner even when they are not. When one of two computing servers is malicious, it could: (B1) send incorrect sum vectors for the highest bid or sale price; (B2) send incorrect AFE shares when finding the winner or the bidder of the sale price.

All malicious behaviors above except (A1) would lead to failure of verification using commitments with high probability due to Pedersen commitments being computationally binding and the hash function being collision resistant. Specifically, after all bidders send a hash of their commitments, they are bound to their AFE vectors. When the random seed used to find the winner and sale price bidder is fixed, the winner and the sale price bidder (in the interactive variant) are also bound. This effectively fixes the outcome of the auction.

(A1) may still pass verification but the outcome of the auction will still be correct since we do not explicitly check whether inputs from all bidders are valid or not. In the non-interactive auction protocol, Addax can treat the highest index with bit one of the decoded bit vector as the bid of the bidder (e.g., [1,0,1,0] corresponds to bid 3). Thus, an invalid AFE vector does not affect the outcome of the auction. And we only need to check the case of (A1) in the interactive variant. If a bidder who submitted an invalid AFE vector does not become the winner or the bidder of the sale price, then they do not affect the auction's outcome. Thus, auditors need only check whether the winner and bidder of the sale price provided valid AFE vectors. We discuss how to do this in Appendix C.

Privacy. We will prove Addax's privacy guarantees using a simulation proof [70]. A simulation proof is done by first defining an ideal functionality $\mathcal{F}$. One can think of it as the function that one would run if one had access to a trusted third party. This ideal functionality will provide some output that is available to everyone, but it will keep all inputs and internal values secret. We want to show that a protocol is as good as the ideal functionality in terms of what information it leaks: anything that an adversary can learn from interacting and observing the output of Addax, the adversary can learn from interacting and observing the output of the ideal functionality. To prove this, we build a simulator *Sim* that interacts with the ideal functionality $\mathcal{F}$ and obtains only the outputs that $\mathcal{F}$ provides without having access to the inputs of the honest parties. If the simulator can produce a *view* (a transcript of all messages sent and received by all parties) that is computationally indistinguishable from the view produced by the execution of the original protocol, we say that the protocol is as secure as the ideal functionality.

To show the security of Addax, we first define a variant that we call Addax-V. This variant differs from the original Addax in that instead of deferring *all* verification to after the protocol finishes, Addax-V verifies the outcome of each computation step (i.e., the highest bid, the winner, and the sale price) immediately after the step completes. If at any step verification fails, the protocol stops without moving forward. Note that we could deploy Addax-V itself, but it would be inefficient; Addax instead moves the verification to the asynchronous step so that it is not part of the critical path of real-time ad auctions.

We will show that Addax-V is as secure as the ideal functionality $\mathcal{F}$; then we will prove that the original Addax protocol with deferred verification is as secure as Addax-V.

Below we give Addax-V's protocol. For simplicity, we omit the exchange of hash values between P_1 and P_2 before sending messages, and whenever Decode-MAX outputs an invalid bit vector, it is assumed that P_1 or P_2 aborts. We also assume that the two auctioneers find the winner (or bidder of sale price) sequentially starting from the first bidder and stopping when the winner (or bidder of sale price) is found.

Addax-V's auction protocol

Step 1 (Bidders encode and send AFE shares):

- Each bidder i among the n bidders encodes its bid as a V-AFE vector M_i, splits it into additive shares M_i^1 and M_i^2, and generates commitments C_i (§5.1).
- Bidder i sends M_i^1 to P_1 and M_i^2 to P_2, and C_i to both P_1 and P_2.

Step 2 (Compute highest bid):

- P_1 sets $s_1 = \sum_{i=1}^{n} M_i^1$; P_2 sets $s_2 = \sum_{i=1}^{n} M_i^2$.
- P_1 and P_2 exchange s_1 and s_2, compute $S = s_1 + s_2$, and run Decode-MAX on S to get b^*.
- P_1 and P_2 use commitments to verify whether b^* is correct (§5.2) and abort if it fails.

Step 3 (Find winner):

For $i = 1$ to n, P_1 and P_2 repeat the following:

- P_1 sends $M_i^1[b^*]$ to P_2, and P_2 sends $M_i^2[b^*]$ to P_1.
- P_1 and P_2 set $\beta_i = \text{Decode-OR}(M_i^1[b^*] + M_i^2[b^*])$.
- If β_i is 1, then i is the winner (set $w = i$); else continue. If $i = n$ and $\beta_i = 0$, then P_1 and P_2 abort.
- P_1 and P_2 ask bidder w if its bid is b^*. If w says no, P_1 and P_2 abort. Else, P_1 and P_2 use commitments to validate w is correct (§5.2) and abort if it fails.

Step 4 (Compute sale price):

- P_1 sends $m_1 = \sum_{i=1}^{n} M_i^1 - M_w^1$ to P_2, and P_2 sends $m_2 = \sum_{i=1}^{n} M_i^2 - M_w^2$ to P_1.
- P_1 and P_2 compute $sp = \text{Decode-MAX}(m_1 + m_2)$.
- P_1 and P_2 use commitments to verify whether sp is correct (§5.2) and abort if it fails.

Note that whenever there is a party that sends *abort*, the protocol terminates and all parties are notified. The detailed process of termination works as follows.

If an auctioneer (either P_1 or P_2) wants to send *abort*, it directly sends *abort* to all bidders and another auctioneer. If a bidder wants to send *abort* when asked whether b^* is its bid, it replies "no" and sends *abort* to the two auctioneers. The two auctioneers then forward the *abort* message to all the bidders.

Now we define an *ideal functionality* that captures the privacy properties of Addax-V's protocol. Let $n = h + k$ be the total number of bidders, where the first h bidders are honest (non-adversarial) and the last k bidders are malicious (actual position is irrelevant). The two auctioneers are denoted as P_1 and P_2. Without loss of generality, we assume an adversary that corrupts P_1 and k bidders. The ideal functionality is:

Ideal functionality $\mathcal{F}$ of Addax-V's auction protocol

Inputs: h bids $b_1, \ldots, b_h$ from h honest bidders and a *cheat* message (an integer from 1 to 4) from P_1.

Outputs: (b^*, w, sp) are computed as:

- $b^* = \max(b_1, \ldots, b_h)$.
- w, such that $b_w = b^*$ while $b_1 \neq b^*, \ldots, b_{w-1} \neq b^*$ (i.e., w is the first bidder whose bid is b^*).
- $sp = \max(b_1, \ldots, b_{w-1}, b_{w+1}, \ldots, b_h)$ (i.e., the maximum bid excluding b_w).

$\mathcal{F}$ conditionally outputs the above computed values depending on the value of *cheat*.

- When *cheat* is 1: $\mathcal{F}$ outputs b^* to P_1 and nothing to P_2.
- When *cheat* is 2: $\mathcal{F}$ outputs (b^*, w) to P_1 and b^* to P_2.
- When *cheat* is 3: $\mathcal{F}$ outputs (b^*, w, sp) to P_1 and (b^*, w) to P_2.
- When *cheat* is 4: $\mathcal{F}$ outputs (b^*, w, sp) to both P_1 and P_2.

Note that the ideal functionality $\mathcal{F}$ also provides an interface to take an *abort* message as input which allows terminating the execution of a protocol by the simulator.

Now we see how the ideal functionality $\mathcal{F}$ captures the privacy properties of Addax-V's protocol. In the real world execution of Addax-V's protocol, there are five different cases: (1) verification in Step 2 fails; (2) verification in Step 2 passes, verification in Step 3 fails, and P_1 does not learn the real winner; (3) verification in Step 2 passes, verification in Step 3 fails, and P_1 learns the real winner; (4) verification in Step 2 and 3 passes but verification in Step 4 fails; and (5) all verification passes.

When *cheat* message to $\mathcal{F}$ is set to 1, the outputs of $\mathcal{F}$ capture cases (1) and (2) above. And when *cheat* is set to other values, the outputs of $\mathcal{F}$ capture the remaining three cases, respectively. Thus, the outputs of $\mathcal{F}$ capture all different cases of real world execution of Addax-V.

Lemma 7. The Addax-V auction protocol securely implements ideal functionality $\mathcal{F}$ under the assumption that the commitment scheme is binding and hiding and that p is large enough to ensure that Decode-OR and Decode-MAX produce incorrect outputs with negligible probability.

Proof. We now build a simulator *Sim* that interacts with the ideal functionality $\mathcal{F}$ which at most leaks the highest bid b^*, the winning bidder's index w, and the sale price sp. Note that the simulator below simulates all the five different cases (when the protocol aborts in different steps) of real world execution in Addax-V. We use $\mathcal{A}$ to denote the adversary who can corrupt P_1 and k malicious bidders. Note that P_1 and P_2 are symmetric and do the same computation in the protocol. Thus, the proof below also applies to an adversary who corrupts P_2.

Simulator *Sim*

Step 1 (Generate random V-AFE vectors):

- $\mathcal{F}$ gives its output to *Sim*. This output depends on which of the five cases of the real world execution we are simulating (based on the *cheat* message).
- For the h honest bidders, *Sim* assigns a bid to each of them in such a way that one of the bids is b^* and all other bidders' bids are set as smaller than or equal b^*.
- If w is included in the output of $\mathcal{F}$, the honest bidder w's bid is the one that is set to b^*.
- If sp is included in the output of $\mathcal{F}$, a random honest bidder's bid (excluding w) is set to sp and all other bids are set as smaller than or equal to sp.
- *Sim* encodes the h honest bidders' bids into h V-AFE vectors and generates commitments (§5). It then splits the V-AFE vectors into additive shares, and sends one of the V-AFE shares and the commitment of each honest bidder to $\mathcal{A}$.
- $\mathcal{A}$ generates V-AFE shares and commitments for the k malicious bidders. It sends one of the V-AFE shares and the commitment of each of the k malicious bidders to *Sim*.
- At this point, $\mathcal{A}$ has shares $M'^1_1, \ldots, M'^1_n$ and *Sim* has shares $M'^2_1, \ldots, M'^2_n$. They both get commitments $C'_1, \ldots, C'_n$.

Step 2 (Compute highest bid):

- *Sim* computes $s'_2 = \sum_{i=1}^n M'^2_i$, and sends it to $\mathcal{A}$.
- $\mathcal{A}$ sends a V-AFE vector s'_1 (s'_1 should be $\sum_{i=1}^n M'^1_i$ if it follows the protocol, or some vector generated based on its cheating strategy) to *Sim*.
- *Sim* computes $b'^* = \text{Decode-MAX}(s'_1 + s'_2)$.
- If $b'^* \neq b^*$, *Sim* sends *abort*.

Step 3 (Find winner):
For $i = 1$ to n, *Sim* and $\mathcal{A}$ repeat:

- *Sim* sends $M'^2_i[b'^*]$ to $\mathcal{A}$, and $\mathcal{A}$ sends v_i (v_i should be $M'^1_i[b'^*]$ if it follows the protocol, or a vector generated based on its cheating strategy) to *Sim*.
- *Sim* computes $\beta'_i = \text{Decode-OR}(M'^2_i[b'^*] + v_i)$.
- If $\beta'_i = 1$, then $w' = i$; else continue to the next round. If $i = n$ and $\beta'_i = 0$, *Sim* sends *abort* to $\mathcal{A}$.
- After finding w', if bidder w' is malicious, *Sim* asks $\mathcal{A}$ whether the bid of w' is b'^*. If $\mathcal{A}$ replies with no, *Sim* sends *abort*.
- If bidder w' is an honest bidder and $w' \neq w$, *Sim* sends *abort* to $\mathcal{A}$.
- If $w' \neq w$, *Sim* sends *abort* to $\mathcal{A}$.

Step 4 (Compute sale price):

- *Sim* sends $m'_2 = \sum_{i=1}^n M'^2_i - M'^2_{w'}$ to $\mathcal{A}$, and $\mathcal{A}$ sends m'_1 (m'_1 should be $\sum_{i=1}^n M'^1_i - M'^1_{w'}$ if it follows the protocol, or a V-AFE vector generated based on its cheating strategy) to *Sim*.
- *Sim* computes $sp' = \text{Decode-Max}(m'_1 + m'_2)$.
- If $sp' \neq sp$, *Sim* sends *abort*.

Note that whenever Decode-MAX outputs an invalid bit vector, we assume that *Sim* sends *abort*. When *Sim* wants to send *abort*, it notifies $\mathcal{A}$ and the ideal functionality $\mathcal{F}$, and $\mathcal{F}$ forwards *abort* and outputs to all honest parties. When $\mathcal{A}$ wants to abort, it sends *abort* to *Sim*, *Sim* forwards *abort* to the ideal functionality $\mathcal{F}$, and $\mathcal{F}$ forwards *abort* and outputs to all honest parties. In the simulation, as long as each party receives one *abort* message, it terminates. Finally, for simplicity whenever $\mathcal{A}$ and *Sim* exchange messages (e.g., V-AFE shares), *Sim* asks $\mathcal{A}$ to send first.

Analyzing the views. When *cheat* is set to 3 or 4 in the ideal world (ideal functionality), which corresponds to the real world (Addax-V) protocol proceeds to the step of computing the sale price, $\mathcal{A}$ learns the entire view in both worlds. The view of $\mathcal{A}$ in the ideal world is: $\{\sum_{i=1}^n M'^2_i, w', b'^*, sp', M'^2_{w'}, M'^2_1[b'^*], \ldots M'^2_{w'}[b'^*], M'^1_1, \ldots, M'^1_n, C'_1, \ldots, C'_n\}$. In the real world (Addax-V), $\mathcal{A}$'s view includes: $\{\sum_{i=1}^{n} M_i^2$, w, b^*, sp, M_w^2, $M_1^2[b^*], \ldots M_w^2[b^*]$, $M_1^1, \ldots, M_n^1$, $C_1, \ldots, C_n\}$.

When *cheat* is set to 1, $\mathcal{A}$'s view in ideal world only includes: $\{\sum_{i=1}^{n} {M'}_i^2$, b'^*, ${M'}_1^1, \ldots, {M'}_n^1$, $C'_1, \ldots, C'_n\}$. In the real world, the corresponding view of $\mathcal{A}$ includes: $\{\sum_{i=1}^{n} M_i^2$, b^*, $M_1^1, \ldots, M_n^1$, $C_1, \ldots, C_n\}$.

When *cheat* is set to 2, $\mathcal{A}$'s view in ideal world only includes: $\{\sum_{i=1}^{n} {M'}_i^2$, w', b'^*, ${M'}_1^2[b'^*], \ldots {M'}_{w'}^2[b'^*]$, ${M'}_1^1, \ldots, {M'}_n^1$, $C'_1, \ldots, C'_n\}$. In the real world, the corresponding view of $\mathcal{A}$ includes: $\{\sum_{i=1}^{n} M_i^2$, w, b^*, $M_1^2[b^*], \ldots M_w^2[b^*]$, $M_1^1, \ldots, M_n^1$, $C_1, \ldots, C_n\}$.

V-AFE shares are uniformly random elements in $\mathbb{Z}_p$, and commitments to V-AFE vectors are hiding, thus they have the same distribution. For w, b^*, sp and w', b'^*, sp', their distributions are also identical. M_w and $M'_{w'}$ are both generated by encoding bid b^* into V-AFE vectors, thus having the same distribution. $\sum_{i=1}^{n} M_i - M_w$ and $\sum_{i=1}^{n} M'_i - M'_{w'}$ are both generated following the requirement that the highest bid among all remaining bidders (excluding bidder w and bidder w') is sp, thus having the same distribution. $M'_1[b'^*], \ldots, M'_{w'-1}[b'^*]$ and $M_1[b^*], \ldots, M_{w-1}[b^*]$ are zeros. These say that in the simulation, bidder 1 to bidder $w' - 1$ are not the winner, and in the real world protocol, bidder 1 to bidder $w - 1$ are not the winner. $M'_{w'}[b'^*]$ and $M_w[b^*]$ are encodings of bit 1.

As a result of the above exhaustive case analysis, both the real world view and the ideal world view are identically distributed. Consequently, an adversary for Addax-V learns nothing beyond what is revealed by the ideal functionality. □

Lemma 8. Addax's protocol does not leak more information to the adversary than the variant Addax-V.

Proof. The only difference between the variant and the original protocol is that in the original protocol, the adversary learns the entire view of $\{\sum_{i=1}^{n} M_i^2, w, M_w^2, M_1^2[b^*], \ldots M_w^2[b^*]$, $M_1^1, \ldots, M_n^1$, $C_1, \ldots, C_n\}$, while in Addax-V, it stops after aborts in Step 2 or 3, and only learns a partial view.

In the original protocol, the malicious auctioneer always learns the correct highest bid regardless of how it behaves, as the honest auctioneer always sends the correct sum of AFE shares. From Lemma 3, if an incorrect bidder is claimed as the winner and the protocol does not abort after finding winner, the incorrect winner w' must be malicious. And in the original protocol, the auctioneers would proceed to compute the sale price with the *incorrect* winner w'. In this case, the AFE vector of the malicious bidder, $M'_{w'}$, is revealed to the auctioneers and auditors.

In Step 4, when computing the sale price, the adversary receives $\sum_{i=1}^{n} {M'}_i^2 - {M'}_{w'}^2$ from the honest party P_2. Adversary knows ${M'}_{w'}^2$ since it's from a malicious bidder, and $\sum_{i=1}^{n} {M'}_i^2$ is already learned in Step 2 to compute the highest bid. Thus, in the original protocol, when the protocol proceeds to Step 4 with an incorrect winner w', it can only learn the same amount of information as what it learned in Step 2 (Lemma 4).

Now we can conclude that the original protocol does not leak more information about *honest* bidders' bids compared to the Addax-V variant, where verification is not deferred. □

C Safeguarding interactive Addax

In the non-interactive protocol, the underlying value of a bit encoding is defined as the rightmost position among all the non-zero values. For instance, both [1,1,1,0] and [1,0,1,0] are the encodings of value 3. The above encoding neither brings issues for privacy nor integrity, but under the interactive variant, this encoding does not work. See an example below.

Suppose a malicious bidder submits an invalid AFE vector which yields an invalid bit vector [1,1,0,1], and all other bidders submit [1,0,0,0]. When finding the winner interactively, two auctioneers will first check the first and the third positions. Since the corresponding results are 1 and 0, the next position to be checked should lie in between the first and the third entry, which, in this example, is the second position. As a result, the highest bid found is 2, which means the bidder wins the auction with bid 2.

Therefore, in the interactive variant, Addax checks whether the AFE vectors of the winner and the bidder of the sale price correspond to valid unary bit encodings. To this end, we add the following two extra steps: (1) an asynchronous procedure for finding sale price bidder (we need this to find *whose* bit encoding to validate); and (2) proving that a bidder's AFE vector is valid without leaking its original AFE vector (we need this to avoid leaking the *third-highest bid*). Note that in the interactive variant of the first-price auction, we also need to prove the winner's AFE vector is valid while hiding its original value so the *second-highest bid* is not leaked.

C.1 Asynchronous: find sale price bidder

The invalid encoding as above impacts the outcome of an auction (if it is not aborted) if the malicious bidder is the winner or the bidder of sale price (if the malicious bidder is neither of these, it has no effect). Therefore, Addax requires validating the AFE vector of the sale price bidder. To this end, Addax has to find its bidder id, though not the real identity. We make the tradeoff of leaking such bidder id in order to keep the completion of the auction within hundreds of milliseconds with this extra asynchronous step.

Specifically, the auxiliary server and publisher first find its bidder id by running Step 4 of Section 4.4 on all AFE vectors except the winning one. This is done *after* the auction is complete and off the latency-critical path (hence why we say this is an *asynchronous* step). Verifiers can check, ex post facto, whether the AFE encodings of the second-highest bidder were valid or not. Similarly to Lemma 3, the bidder found in this step is either the real bidder of the sale price or a malicious bidder.

C.2 Checking the validity of a V-AFE vector

Insecure strawman. To check the validity of V-AFE vectors we can let the vectors be revealed in the clear and check the validity by ensuring they are in the right unary bit form, and are consistent with the Pedersen commitments. However, if we do this for the second-highest bidder's V-AFE vector, this would result in leaking the third-highest bid—substracting the V-AFE vectors of the winner and sale price bidder from the overall sum vector (**M**) reveals the sum of V-AFE vectors of the remaining bidders, thus leaking their maximum bid. We provide the following construction to check validity of the V-AFE vector of the sale price bidder without leakage.

Secure check for AFE validity. In a nutshell, the idea is that in Step 1 of the auction protocol (§ 4.4), bidders additionally generate a V-AFE vector **vr** with a random mask **r** for their original V-AFE vector **v**, such that we can still check the validity of **v** by utilizing **vr**.

Specifically, the bidder first generates the random mask **r**, which is a vector of ℓ random non-zero elements in $\mathbb{Z}_p$ and **vr** is the element-wise product between the two $\{r_1 \cdot v_1, \ldots, r_\ell \cdot v_\ell\}$. Since $r_i \cdot v_i = 0$ if and only if $v_i = 0$ (for $1 \le i \le \ell$), therefore, as long as **vr** is decoded to be a valid bit vector, so is **v**.

When generating commitments for V-AFE vector **v**, a bidder also generates commitments to **vr** and a proof that the underlying messages in the above two commitments indeed differ by a multiplicative factor **r**. Concretely, the bidder uses a *Sigma protocol* [78], which is type of very simple and efficient zero-knowledge proof (with the *Fiat-Shamir heuristic* [53] it is made into a non-interactive proof) to prove that, for each element cr_i in the commitments of **vr** and c_i in commitments of **v**, there exists a non-zero r_i which satisfies $cr_i = c_i^{r_i}$. We give details about the above zero-knowledge proof in Appendix C.4. Appendix C.3 proves the binding and hiding properties of the commitment to **vr** (which is slightly different than standard Perdersen commitments).

During verification, the sale price bidder reveals only its **vr**, and an auditors: (1) verify whether **vr** is consistent with its commitment; (2) check that the decoded result of **vr** is a valid AFE encoding; (3) verify the zero-knowledge proof with respect to the commitments of **v** and **vr**.

A key optimization to this process is as follows. Observe that in the interactive variant the two auctioneers only compute the sum vector of partial entries (e.g., they may only use the first 100 entries to compute based on the lower/upper bounds derived). Thus, the sale price bidder need only hide its original AFE vector for those entries (by using the above **vr** and zero-knowledge proofs for those entries); the sale price bidder can actually reveal its original AFE encoding for the other entries without need for proofs. The result is that auditors need only check the zero-knowledge proofs for the partial entries used to compute the sale price.

Remark. First, the above approach only hides non-zero values in **v**, (i.e., it leaks whether a value in **v** is zero or not). This is because for entries with zero values in the V-AFE vector of sale price bidder, those entries in the sum vector are also zeros, as the sum vector computed in Step 5 (§4.4) decodes to the bid of the sale price bidder. Thus, learning those entries with zero values does not leak the bids of any other bidders.

Second, an adversary may use zeros in the random mask **r** to flip non-zero values into zero, which might turn an invalid AFE vector into a valid one. To check that a bidder did not use zero as random mask, for each tuple vr_i in **vr**, we check that the second element of vr_i is not zero.

C.3 Commitment to a V-AFE tuple with random mask

Given a V-AFE tuple $v = (a, b)$, its tuple with a random mask r is $vr = (r \cdot a, r \cdot b)$. The commitment is as follows. Let $\mathbb{G}$ be a group of prime order p and let $\{g, h\}$ be two random generators $\{g, h\}$ of $\mathbb{G}$. The commitment is $g^{r \cdot a} \cdot h^{r \cdot b}$.

The reason why the above is slightly different than standard Pedersen commitments is that the randomness (the exponent of h in Pedersen) is sampled uniformly at random and independent of the exponent of g, whereas here there is a relationship between the exponent of g (which is $r \cdot a$) and the exponent of h (which is $r \cdot b$).

Lemma 9. Let $c = g^{r \cdot a} \cdot h^{r \cdot b}$ be a commitment to $vr = (r \cdot a, r \cdot b)$. Then c perfectly hides vr, and computationally binds vr if the discrete logarithm problem is hard in $\mathbb{G}$.

Perfect hiding. The commitment perfectly hides both r and a. Let $x \in Z_p$ be an element such that $g = h^x$ (this is well defined since h is a generator and hence there exists an x such that $h^x = g$). Given r, a, b, for any a' there exist r' and b' such that $g^{r \cdot a} \cdot h^{r \cdot b} = g^{r' \cdot a'} \cdot h^{r' \cdot b'}$. And r', b' satisfy that $r' = \frac{x \cdot r \cdot a + r \cdot b}{x \cdot a' + b'}$. Thus, the commitment hides a. Using a similar proof, we can show that the commitment also hides r.

Binding. We now prove that for a message $m = r \cdot a$, the commitment $g^{r \cdot a} \cdot h^{r \cdot b}$ binds m. Specifically, our goal is to show that, if an adversary can find a different message $m' = r' \cdot a'$ and some randomness b' such that $g^{r \cdot a} \cdot h^{r \cdot b} = g^{r' \cdot a'} \cdot h^{r' \cdot b'}$, then it can break discrete log for the instance $(g, h = g^x)$. Note here we only assume $m' \neq m$, and it does not mean $r' \neq r$ or $a' \neq a$.

Suppose the adversary finds such r', a', b' as above, which implies $r \cdot a + r \cdot b \cdot x = r' \cdot a' + r' \cdot b' \cdot x$ where $h = g^x$ for some x (i.e., $r \cdot a - r' \cdot a' = (r' \cdot b' - r \cdot b) \cdot x$). If $r' \cdot b' = r \cdot b$, then the above equation means $r \cdot a = r' \cdot a'$, which contradicts with our assumption that $m' \neq m$. If $r' \cdot b' \neq r \cdot b$, then the adversary can compute $x = (r' \cdot b' - r \cdot b)^{-1}(r \cdot a - r' \cdot a')$, which means now the adversary solves discrete log for instance $(g, h = g^x)$.

C.4 Zero-knowledge proof of commitment relation

The prover, which is the bidder in our case, wants to prove that the there exists a secret element $r \in \mathbb{Z}_p$ such that commitments cr and c satisfy $cr = c^r$. Figure 8 gives the pseudocode for how the prover generates the non-interactive proof π. The prover first samples a random element r' from $\mathbb{Z}_p$, and computes $u =$

Prover$(c, r, cr = c^r)$	**Verifier**$(c, cr = c^r, \pi = (u, v, z))$
$r' \xleftarrow{R} \mathbb{Z}_p$	$v \overset{?}{=} H(c, cr, u)$
$u \leftarrow c^{r'}$	$c^z \overset{?}{=} u \cdot cr^v$
$v \leftarrow H(c, cr, u)$	
$z \leftarrow r' + v \cdot r$	
output $\pi = (u, v, z)$	

FIGURE 8—Non-interactive zero-knowledge proof of knowledge of secret r using the Fiat-Shamir Heuristic. H is a random oracle and its range is $\mathbb{Z}_p$ (heuristically instantiated with a secure hash function).

$c^{r'}$. The prover then computes $H(c, cr, u)$ as the challenge v. H is modeled as a random oracle but heuristically instantiated with a collision-resistant hash function. Finally, the prover computes $z = r' + v \cdot r$. The prover sends $\pi = (u, v, z)$ to the verifier, and the verifier checks π as described in Figure 8.

C.5 Proofs of lemmas 5 and 6

Below we prove the Lemmas for the interactive variant of Addax which leverages the safeguards described in this section.

Proof of Lemma 5. The non-interactive protocol and interactive variant differ in the following two places: (1) in interactive variant, computing MAX only uses partial entries; (2) in interactive variant, Addax checks validity of winner and sale price bidder's AFE vectors with additional verification materials (Appendix C.2).

For (1), in the non-interactive protocol, adversary learns the entire sum vector $\mathbf{M}$, while in the interactive variant, it only learns $r \cdot \lceil \ell^{1/r} \rceil$ entries of $\mathbf{M}$ (§6.1). It therefore leaks no more information about bids than the non-interactive protocol.

For (2), in the interactive protocol, the winner w reveals its full AFE vector, and the sale price bidder needs to provide materials as in Appendix C.2. In the non-interactive protocol, w's full AFE vector can already be inferred from Step 3 (which computes $\mathbf{M}$) and Step 5 (which computes $\mathbf{M} - M_w$), so this leaks no additional information. And additional materials provided in the interactive variant leak no more information of bids than the sale price as detailed in Appendix C.2.

Proof of Lemma 6. Integrity is ensured in the non-interactive protocol as per Theorem 1. The only difference in the interactive variant is that if either the winner or the sale price bidder submits an invalid AFE vector, it can lead to the k-ary search in the interactive protocol to converge to an incorrect value. As we discuss in Appendix C, Addax adds checks to ensure that the AFE vectors of the winner and sale price bidder are both correct, and hence the k-ary search converges to the same value as in the non-interactive protocol.

D Subsets of faulty parties

An auction may be aborted during the online phase, or deferred verification may fail due to a lack of enough materials on the public log (e.g., commitments, sum shares), or due to inconsistent materials such as the bidder sending invalid shares to auctioneers, or an auctioneer claiming it received one value when a bidder submitted another. Figure 9 gives pseudocode for how Addax narrows down the parties at fault. Below is a text explanation for the pseudocode.

Auction aborts. An auction may abort for two reasons: (1) Decode-MAX outputs an invalid bit vector; or (2) the chosen winner or sale price bidder claims that their bids do not equal the found highest bid or sale price. For case (1), if verification on the decoded sum vectors passes, then it implies that some bidders provided invalid AFE vectors. Addax assigns blame to all participating bidders as potentially malicious. If verification of the sum vectors fails, Addax assigns blame to all bidders, the publisher, and the auxiliary server. For case (2), auditors check the shares revealed by the publisher and auxiliary server, and check whether they are consistent with the corresponding commitments. If they are not consistent, Addax assigns blame to the specific bidder (winner or second highest bidder), publisher, and auxiliary server. If it is consistent, and that entry decoded to 0, Addax assigns blame to the auxiliary server and the publisher; if the entry decoded to 1, Addax assigns blame only to the corresponding bidder.

Lack of materials. Participants may not upload all required materials to the public log, which prevents auditors from verifying the auction. Bidders are bound to their bidder ids which should be uploaded by the publisher and auxiliary server. An auditor can easily tell who did not upload the required materials and assign blame to that set of participants.

Inconsistent views between publisher and auxiliary server. The publisher and auxiliary server may provide an inconsistent view for messages they send and receive. For example, the publisher may claim that it receives sum vector M from the auxiliary server, while the auxiliary server claims that it sends M' to the publisher. In this case, Addax assigns blame to both publisher and auxiliary server. If publisher and auxiliary server upload different views of hash values of certain bidder, Addax assigns blame to the publisher, auxiliary server, and the specific bidder, as the bidder may send inconsistent hash values on purpose.

Inconsistency between hash values and commitments. Auditors may find that hash values of commitments uploaded by publisher and auxiliary server are inconsistent with that uploaded by the bidder. Addax assigns the blame to that particular bidder (since publisher and auxiliary server are assumed to not collude).

Inconsistent AFE sum vectors or bit encodings. Verification on sum vectors or revealed bit encodings to find the winner may fail. If verification on sum vectors fails, Addax assigns blame to publisher, auxiliary server and all bidders. If verification on specific bidder's bit encoding fails, Addax assigns blame to publisher, auxiliary server and the specific bidder.

Invalid AFE vector. The winner needs to upload its full AFE vector and the bidder of sale price needs to upload its full AFE

```
 1: function AssignBlame($materials, abort, auctioneers, bidders$)
 2:     # Check if abort happens
 3:     if $abort \neq null$ then
 4:         if $abort.decodeMax == true$ then
 5:             if $verifySumvec(materials) == true$ then
 6:                 $blame(bidders)$
 7:             else
 8:                 $blame(auctioneers, bidders)$
 9:         else if $abort.findBidder == true$ then
10:             if $verifyBit(materials.bidders[abortId]) == false$ then
11:                 $blame(auctioneers, bidders[abortId])$
12:             else
13:                 if $decode(materials.bidders[abortId].bitEncoding) == 0$ then
14:                     $blame(auctioneers)$
15:                 else
16:                     $blame(bidders[abortId])$
17:     # Check all materials are not missing
18:     for $auc$ in $auctioneers$ do
19:         if $materials.auc == null$ then
20:             $blame(auc)$
21:     for $b$ in $bidders$ do
22:         if $materials.b == null$ then
23:             $blame(b)$
24:     # Check inconsistency between auctioneers
25:     if $inconsistent(materials.auctioneers.sumvec)$ then
26:         $blame(auctioneers)$
27:     for $b$ in $bidders$ do
28:         if $inconsistent(materials.auctioneers.hash[b])$ then
29:             $blame(auctioneers, b)$
30:     # Check inconsistency between hash and commitments
31:     for $b$ in $bidders$ do
32:         if $inconsistent(materials.b.hash, materials.b.commitment)$ then
33:             $blame(b)$
34:     # Verify sum vectors and bit encodings
35:     if $verifySumvec(materials) == false$ then
36:         $blame(auctioneers, bidders)$
37:     for $b$ in $bidders$ do
38:         if $verifyBitEncoding(b.materials) == false$ then
39:             $blame(auctioneers, b)$
40:     # Validate AFE vector of winner and sale price bidder
41:     if $validate(materials.bidder.AFE) == false$ then
42:         $blame(bidder)$
```

FIGURE 9—Pseudocode of how to assign blames in Addax, see texts for more details.

vector with random mask (§C.2), the commitments and non-interactive zero-knowledge proofs. An auditor needs to check the winner's AFE vector decodes to be a valid bit vector. Also, an auditor must check whether the AFE vector with random mask is consistent with the supplied commitments, whether it decodes to be a valid bit vector, and verify the proofs. If the check fails, Addax assigns blame to the specific bidder. If the check passes, even if verification on the sum vectors fails, Addax explicitly knows that these two bidders are honest, and can avoid assigning blame to them. Addax will then protect their identities.

D.1 Narrow down faulty bidders when both auctioneers are honest

There are some cases (e.g., line 8 in Figure 9) where Addax can only assign blame to all of the bidders due to the fact that a malicious auctioneer could collude with bidders. However, if both auctioneers were honest (how one would establish this is orthogonal, though likely hard), Addax can identify the specific bidders who provided inconsistent AFE vectors and commitments. To detect such faulty bidders, each of the two auctioneers computes the commitment over its local V-AFE vector share of a bidder and reveals the commitment. They then verify whether multiplying these two commitments from both auctioneers yields the commitment submitted by the bidder. If not, then Addax can blame the bidder.

E Interacting with the public log

E.1 A brief primer on Algorand

Accounts and Transactions. An account in Algorand consists of a key pair. Transactions include payment, key registration, and asset transferring. Each transaction is created by one account and must be signed with its corresponding secret key.

Smart contracts and application calls. Smart contracts are programs that run on the blockchain with user-defined functionality. Each smart contract is specified with a unique ID. Application calls are transactions used to invoke functions in smart contracts like RPC calls. To interact with a smart contract, an account needs to join the smart contract first. The `opt-in` call allows one account to join one smart contract instance while a `close-out` call allows one account to leave. Addax maintains one smart contract per ad category which includes all advertisers' information in that category. When one advertiser belongs to multiple categories, it joins all related smart contracts. There is no upper bound on the number of accounts that can join one smart contract in Algorand. Bidders invoke application calls defined in the running smart contract to insert, update, or delete their own information.

Indexer. Indexers are special nodes which provide RESTful interfaces to search for transactions or states of certain apps by answering SQL-like queries. For example, it can answer queries to search for all transactions that happened during a certain period in one smart contract instance with a query like: `SELECT * from transactions WHERE appID = {ID of App} AND after-time = {start} AND before-time = {end};`

E.2 Workflow of a deployed smart contract

There are two kinds of smart contracts in Algorand, stateful ones which have their own storage and stateless ones which do not. Storage in smart contracts consists of several key-value pairs which can be read and written. There is *global storage* which maintains the state of the smart contract instance and *local storage*. All opted-in accounts have their own local storage. All storage can be read by anyone and updated via application calls. Application calls for writing local storage can only be invoked by its owner account.

Addax maintains one stateful smart contract per ad category which includes all advertisers in that category. When one advertiser belongs to multiple categories, it joins all related

```
1: function INIT(N)
2:     $gs.counter \leftarrow 0$
3: function CREATE(info)
4:     $ls.id \leftarrow gs.counter$
5:     $gs.counter \leftarrow gs.counter + 1$
6:     $ls.info \leftarrow info$
7: function UPDATE(newInfo)
8:     $ls.info \leftarrow newInfo$
9: function DELETE
10:    $delete(ls.info, ls.id)$
```

FIGURE 10—Pseudocode of smart contract, *gs* is global state, *ls* is local storage of each advertiser. Init function is called when smart contract is initialized. Create, Update, Delete functions are called by advertisers.

smart contracts. Figure 10 shows the functionalities provided by the deployed smart contracts in Addax. Each smart contract is created by an `INIT` function to initialize an incremental counter starting from zero in global storage. Advertisers can invoke `CREATE`, `UPDATE` and `DELETE` functions. `CREATE` is the `opt-in` application call in Addax that opts an advertiser into the smart contract of its category and write information into the invoker's local storage. Local information of advertisers can include brand name, all categories the advertiser belongs to, domain and port of ad server, protocols and serialization formats supported, etc. The `UPDATE` call is invoked by opted-in advertisers to update their local information. Advertiser invokes `DELETE` to clear all information stored in local storage and leave this smart contract instance.

E.3 Costs of interacting with public log

In this section, we answer the question of costs for interacting with the public log and for querying the indexer. We use the PureStake indexer which provides a REST API that one can use to upload and retrieve information from the Algorand blockchain. PureStake has servers all over the world as they contract with Cloudfront. In our experiments, requests to PureStake that originate from AWS US East (Ohio) contact servers in Ontario. Requests that originate from the AWS US West (California) contact servers in California. Requests that originate from AWS US West (Oregon) contact servers in Oregon. PureStake then takes care of broadcasting the transaction to the Algorand peer-to-peer network.

The size of advertisers' information (§E.1) uploaded to Algorand to participate in Addax is 960 bytes. We experiment with a modest number of advertisers. The reason that we do not have tens of thousands of advertisers is that creating a new advertiser requires creating an Algorand account (new email address, password, account verification, etc.) which is time-consuming. Nevertheless, we semi-automate this painstaking process and generate 1,000 accounts.

Time for advertisers' operations. In Figure 11, we evaluate the time of advertisers' operations (invoking CREATE, UPDATE or DELETE application call) on the Algorand blockchain. We vary whether advertisers belong to one or multiple categories, and also vary the distribution of the number of opted-in advertisers for a given category. Figure 11a shows the results of an advertiser interacting with Algorand where the advertiser belongs to a single category while varying the number of advertisers registered for that category. Figure 11b shows the results when the advertiser belongs to multiple categories, all of which have 500 advertisers registered. Finally, Figure 11c shows the results when the advertiser belongs to six different categories, while each category contains a different number of advertisers. In Scenario 1, each category contains 100 advertisers. In Scenario 2–4, the numbers of advertisers in each category are [50, 100, 100, 100, 100, 150], [50, 50, 50, 100, 150, 200], and [50, 50, 50, 100, 100, 250].

As can be seen, the time for these three operations remains nearly constant when the number of opted-in advertisers in one category grows. It takes about 8–8.5 seconds for invoking one application call to one category (i.e., one smart contract instance). Most of the overhead (about 7 seconds) comes from advertisers waiting for confirmation from the blockchain that this operation has finished successfully. Advertisers can directly invoke application calls and leave without having to wait for confirmation. Indeed, this is what the browser does when uploading its audit materials as described in the Leaving an audit trail paragraph of Section 9.4.

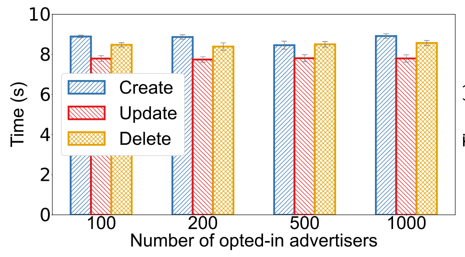
The costs of these operations grow linearly with the number of categories to which an advertiser belongs, regardless of the number of opted-in advertisers in each category.

Time for querying the indexer. We also evaluate the costs for querying the indexer for updates during certain period of time under different scenarios. The time for querying the indexer is also the time to verify the query results from cache servers. Figure 12a shows the time to query indexer for updates of multiple categories. Each category contains 500 advertisers. We simulate 20% updates in each category, namely 100 transactions happened during the period we search for. Figure 12b shows the time to query the indexer for updates of one category with 1,000 advertisers but with different percentages of updates during the period of search.

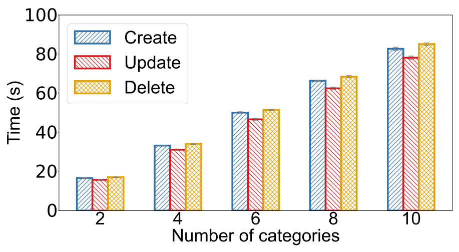
We find that querying one category generally takes about 0.8 seconds, and this number would grow slightly if the number of total updates (i.e., transactions) grows. This is due to the fact that as the total number of updates grows, the data fetched from the indexer grows as well. Also, the time to query the indexer for updates of multiple categories grows linearly with the number of categories queried. This is because to query *N* categories, the querier needs to send *N* requests to the indexer.

F What about TEE-based solutions

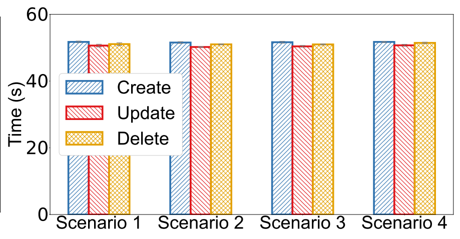
In principle, one might be able to design a solution that leverages TEEs to provide privacy and public verifiability for online ad auctions. However, this is not a trivial task, since TEEs:



(a)

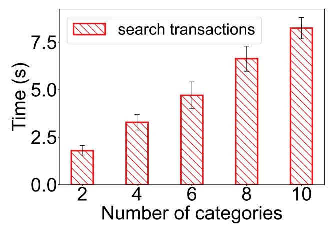


(b)

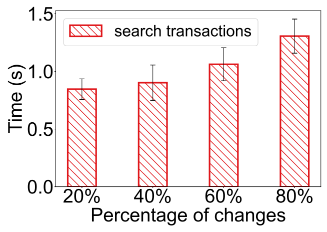


(c)

FIGURE 11—Time for advertisers' operations on the Algorand blockchain under different settings (see text).



(a)



(b)

FIGURE 12—Time to query indexer for updates in certain period under different scenarios.

- Require the release of code that runs inside the enclave. This includes the auction protocol, the encryption/decryption code to recover plaintext bids, and signing code for creating a proof that can be publicly audited.
- Require careful auditing or formal verification of all the code running inside the enclave to ensure the exchange operator (who is running the TEE) did not inject backdoors or other vulnerabilities that can obviate the TEE.
- Intel SGX in particular requires trusting an Intel cloud server during remote attestation (cloud services like Azure's Attestation Service [29] can also be used). In either case, trusting such servers might not be that different of an assumption than the anytrust model in Addax.
- Require additional mechanisms to prevent replay attacks. For example, suppose an operator runs an auction, invites 10 bidders, and passes as input their 10 encrypted bids to TEE (TEE internally has a key to decrypt bids). The TEE then outputs the winner and sale price in the clear. The operator could then run the same auction again but passing only a subset of the bids (these are all valid encrypted bids under a key known to the TEE). The TEE then outputs a winner and sale price in the clear, so the attacker could quickly discover all bids. In contrast, replay is not possible in Addax since the auction is either completed so both auctioneers forward the result to the publisher, or aborted so at least one honest auctioneer forwards an abort result to the publisher (and the publisher displays a generic ad).

G Compatible user privacy features

One of Addax's goals is to have a flexibile enough design to be compatible with various efforts that aim to improve user-privacy (these efforts are orthogonal but they are also complementary to Addax).

Bidding on groups rather than individuals. User activities are tracked as advertisers need to gather enough information about different users' browsing and purchasing preferences. The information is used by advertisers to decide how to bid for a user. Addax's design is compatible with Google's recent Topics proposal [16] which locally groups users into groups. In particular, Topics enhances the browser to keep track of the user's interest and assigns to the user a Topic identifier. Once this identifier exist, Addax can send this identifier to the selected bidders instead of sending them more demographic information (§7). An advertiser would then decide how to bid for each group of users without learning information about the individual user for which it is bidding. The obvious caveat here is that the current Topics proposal is not perfect and there have been various privacy concerns voiced [5, 6, 17].

Measuring conversions without learning individual's data. Measuring the effectiveness of an ad after the auction is essential for advertisers. However, the current way of measuring conversions leaks users' sensitive information about which websites are visited. There is a recent effort [34] that provides a mechanism to measure the return on investment (ROI) and conversions without requiring the advertiser to learn information about a specific user. At the end of the measurement, advertisers see a differentially private histogram of all users' conversions, which is sufficient for them to determine the effectiveness of their campaigns. In Addax, after the auction finishes, advertisers could apply this approach to privately gather data about conversions for analysis while being more sensible to users' privacy concerns. This mechanism is compatible with Addax as it occurs *after* the auction completes.